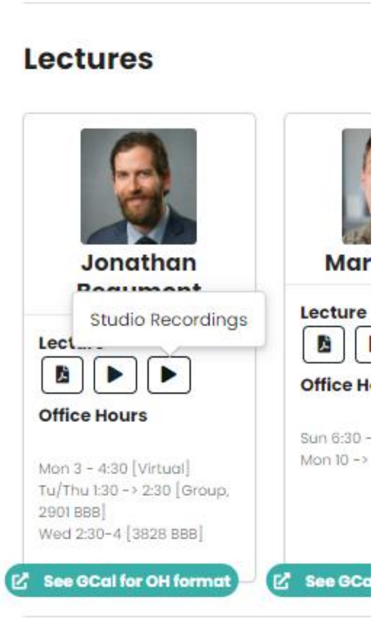
EECS 370 - Lecture 11

Multi-Cycle Data Path

Reminder

- If you're watching lectures asynchronously…
- I have studio recordings
  - Much better quality than lecture recordings
  - I won't walk off screen, etc



Mon 3 - 4:30 [Virtual]
Tu/Thu 1:30 -> 2:30 [Group, 2901 BBB]
Wed 2:30-4 [3828 BBB]

Sun 6:30 -
Mon 10 ->

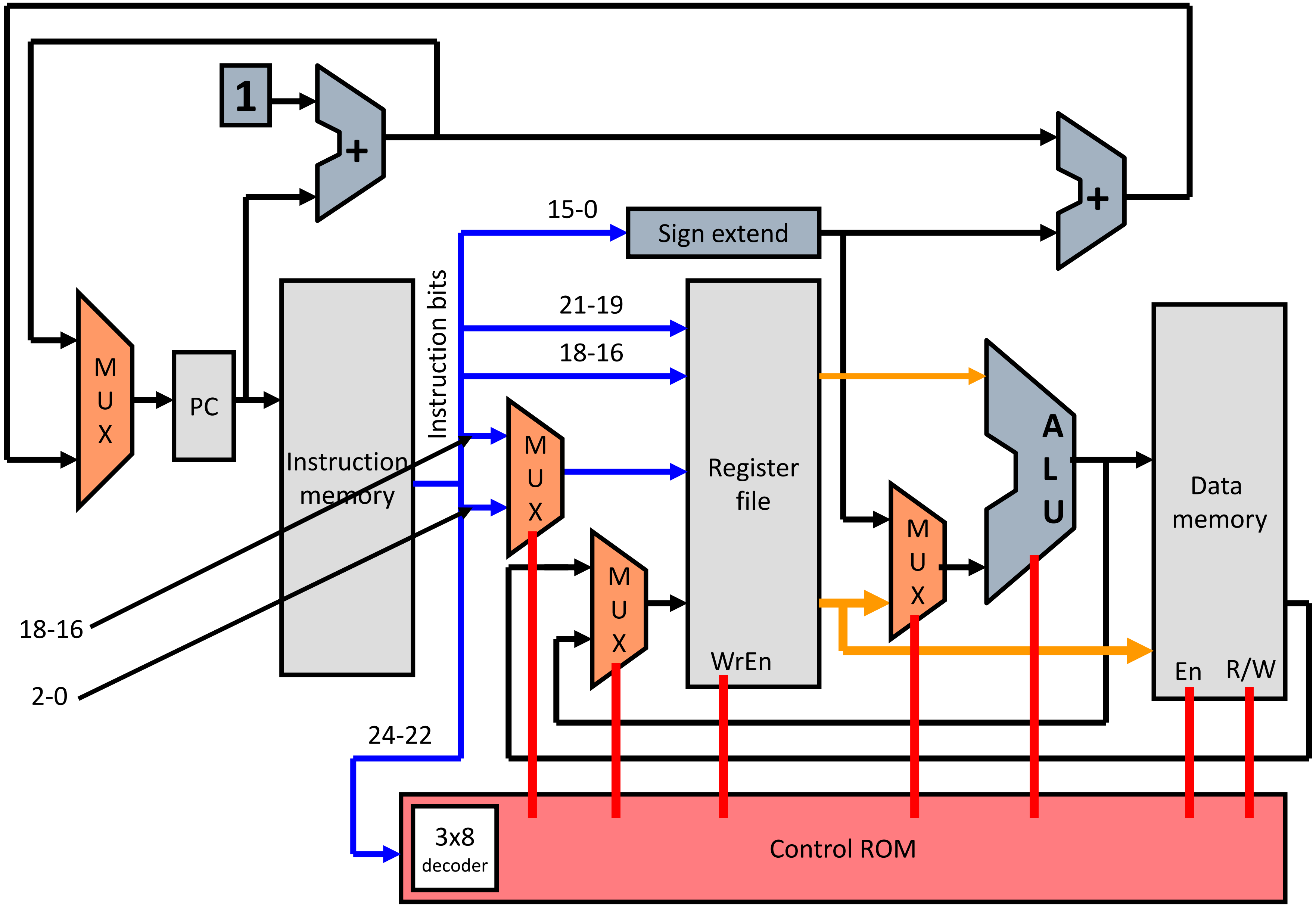
Extra Lecture Section

- Professor Gokul has moved his lecture section to 10:30 am in EWRE
  - Feel free to attend those if that time works better for you

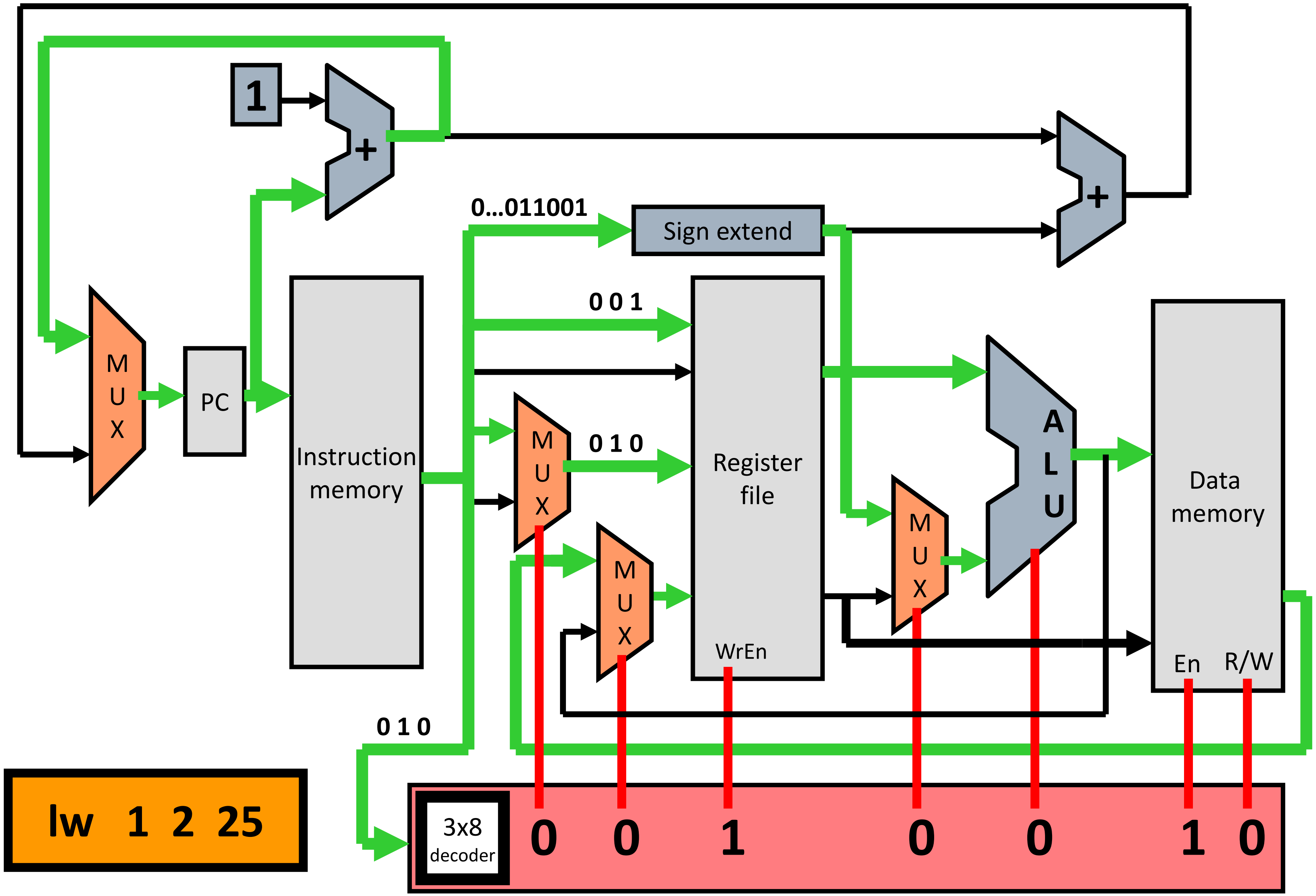
Announcements

- P2
  - P2a due Thursday
  - P2L due in a month
- HW 2
  - Posted, due next week
- Lab 5 due Wed
- Midterm Exam
  - Thu Oct 12th, 6-8 pm (next week)
  - Sample exams on website
  - You can bring 1 sheet (double sided is fine) of notes
  - We will provide LC2K encodings + ARM cheat sheet

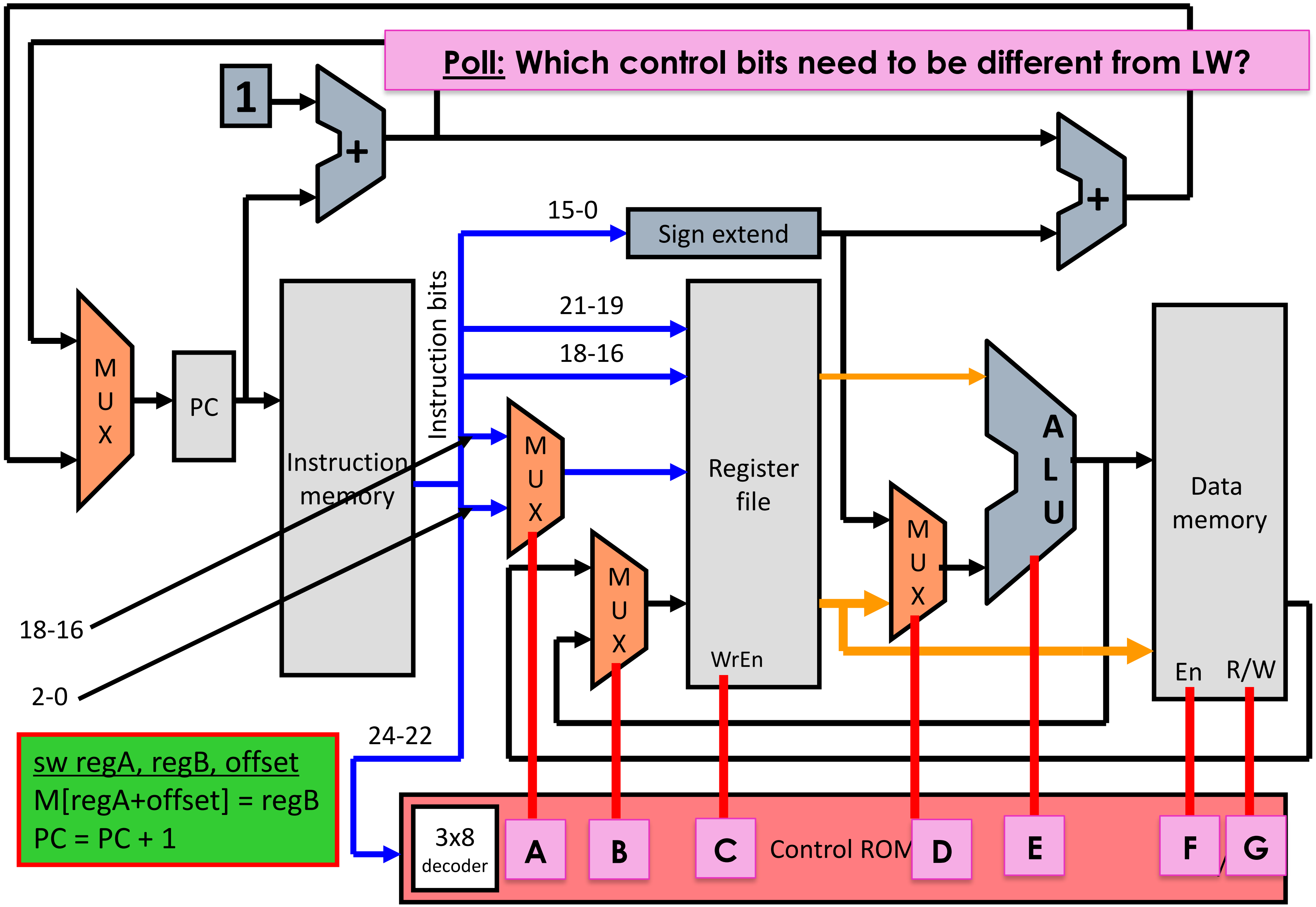
LC2K Single-Cycle Datapath Implementation



Executing a LW Instruction on LC2Kx Datapath



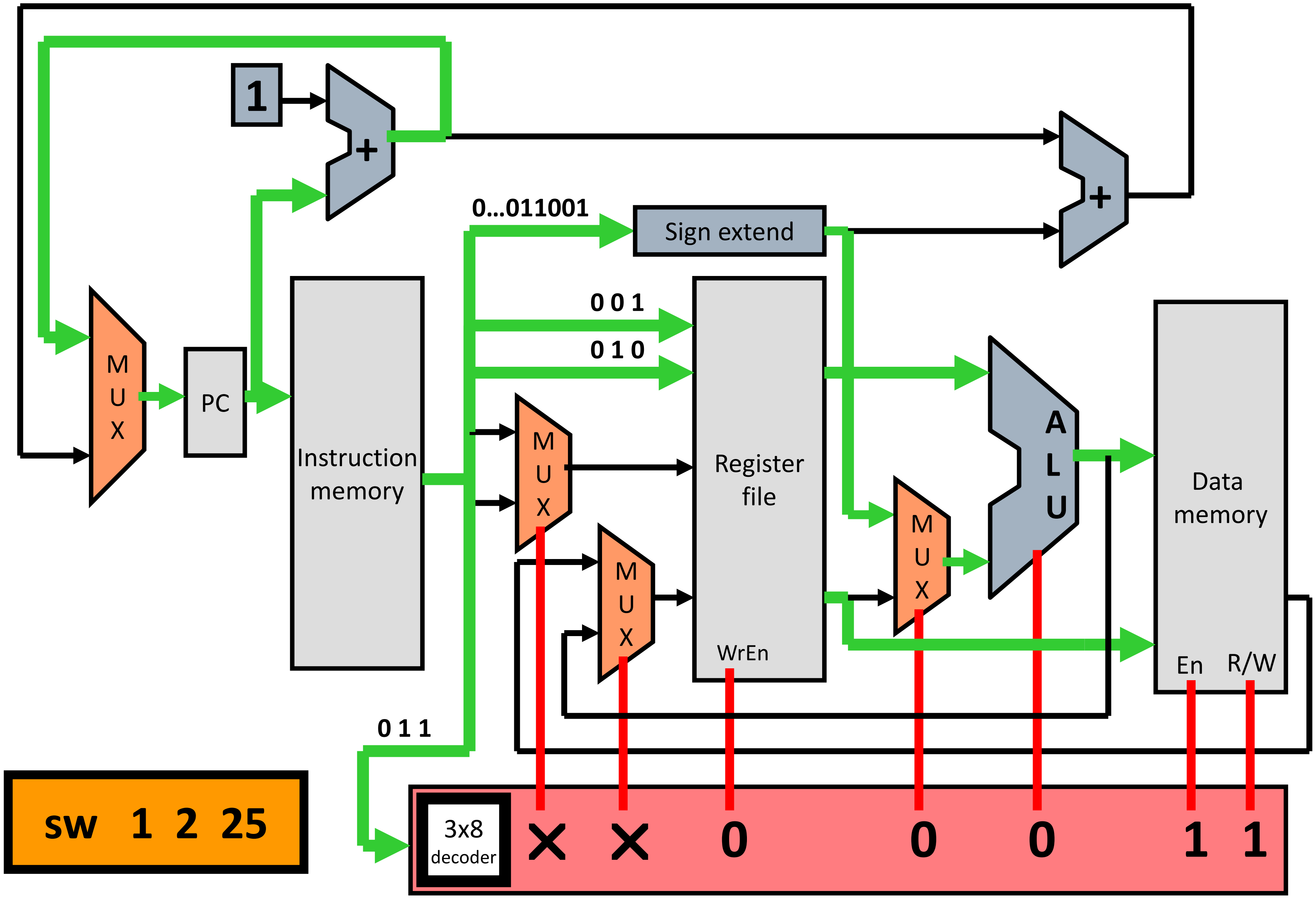
Executing a SW Instruction



Poll: Which control bits need to be different from LW?

sw regA, regB, offset
M[regA+offset] = regB
PC = PC + 1

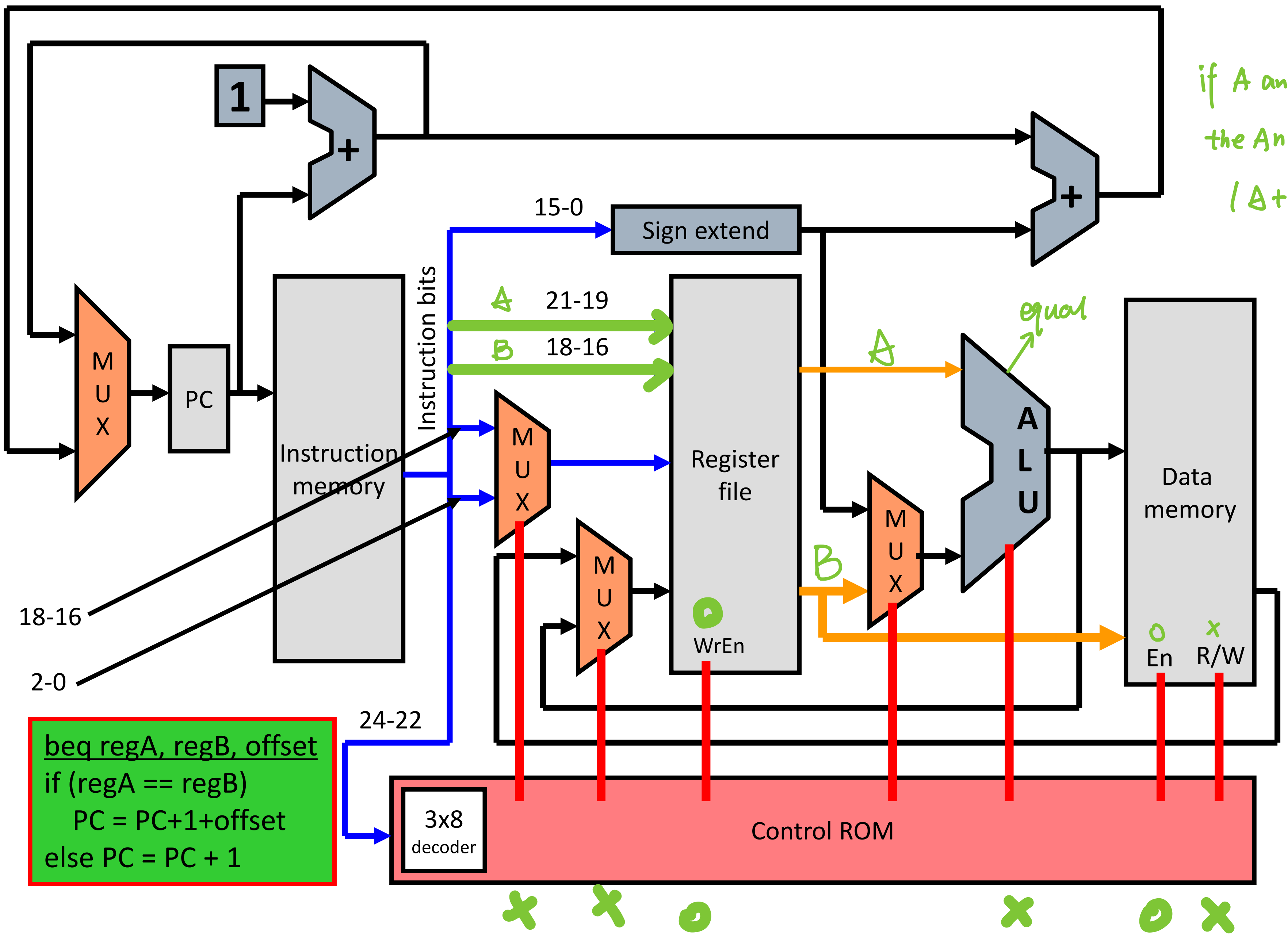
Executing a SW Instruction on LC2Kx Datapath



Executing a BEQ Instruction

if regA = regB. ⟶ Branch to PC+1+offset.

if regA ≠ regB. ⟶ Branch to PC+1.



beq regA, regB, offset

if (regA == regB)

PC = PC+1+offset

else PC = PC + 1

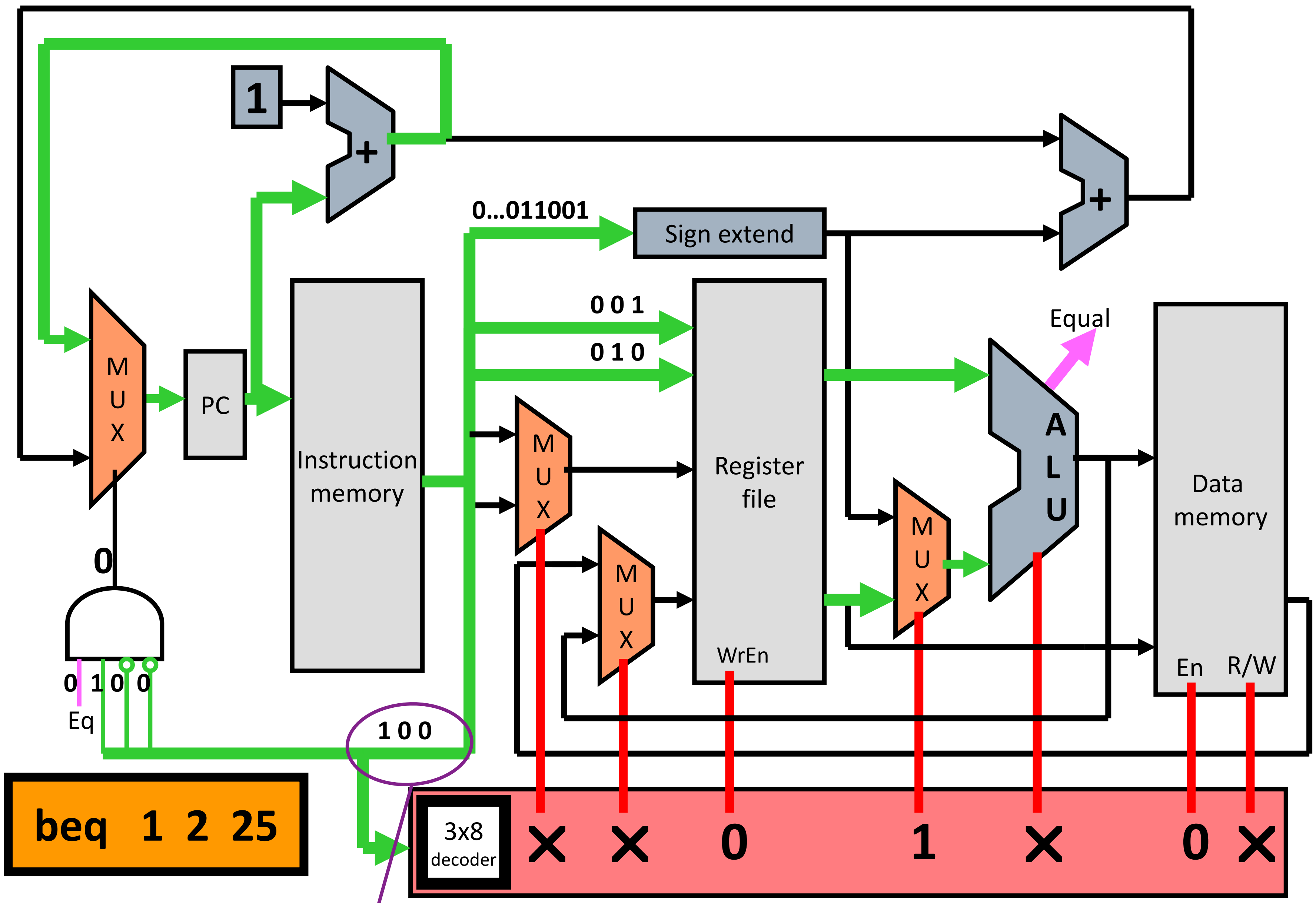
if A and B are the same

the A nor B =

$(A+B)'$ $A'B'$

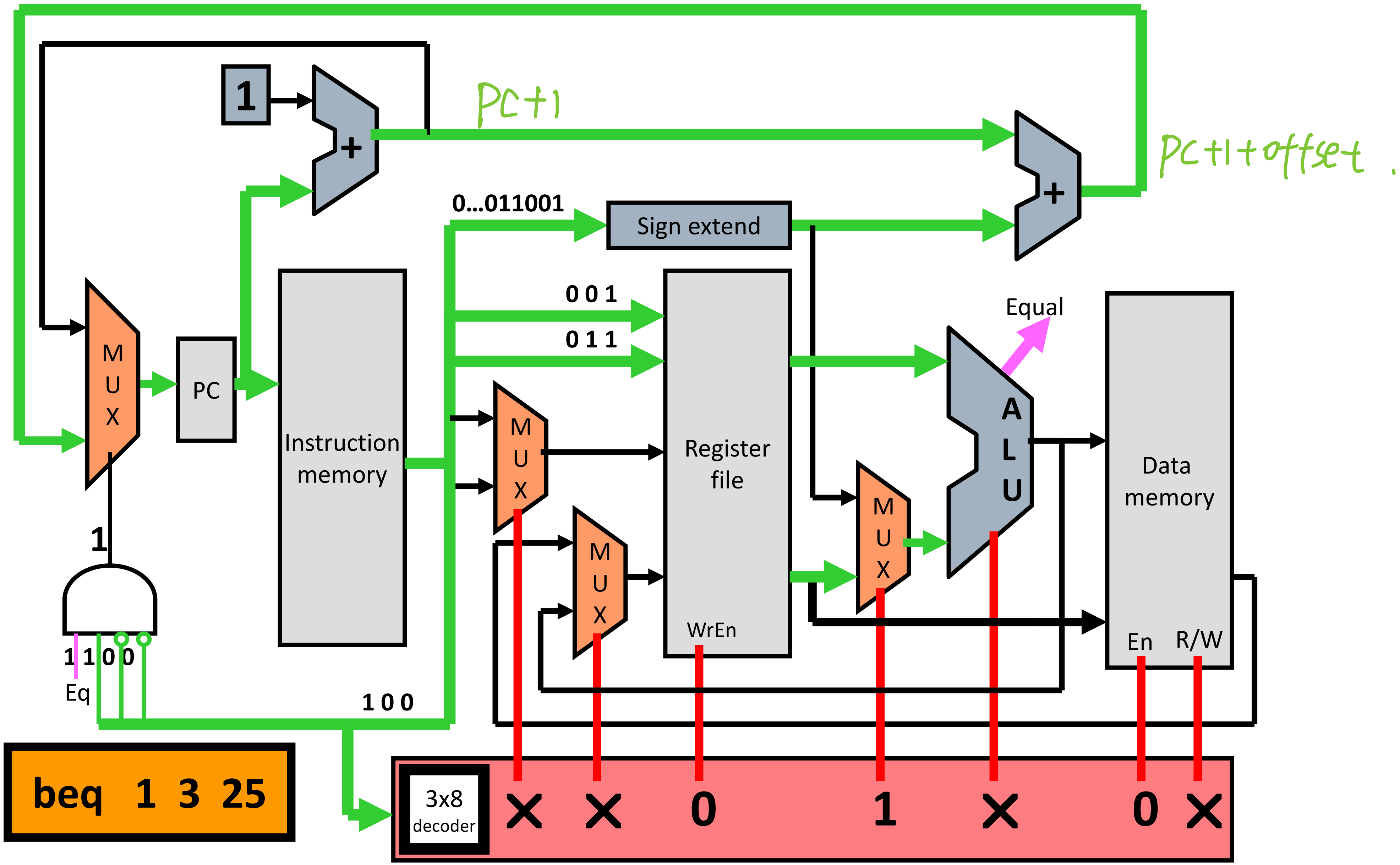
A	B	
0	0	1
1	0	0
0	1	0
1	1	0

Executing “not taken” BEQ Instruction on LC2K Datapath



opcode的信息不足以用于判断是 PC=PC+1 还是PC=PC+1+offset.

Executing a “taken” BEQ Instruction on LC2K Datapath



beq 1 3 25

So Far, So Good

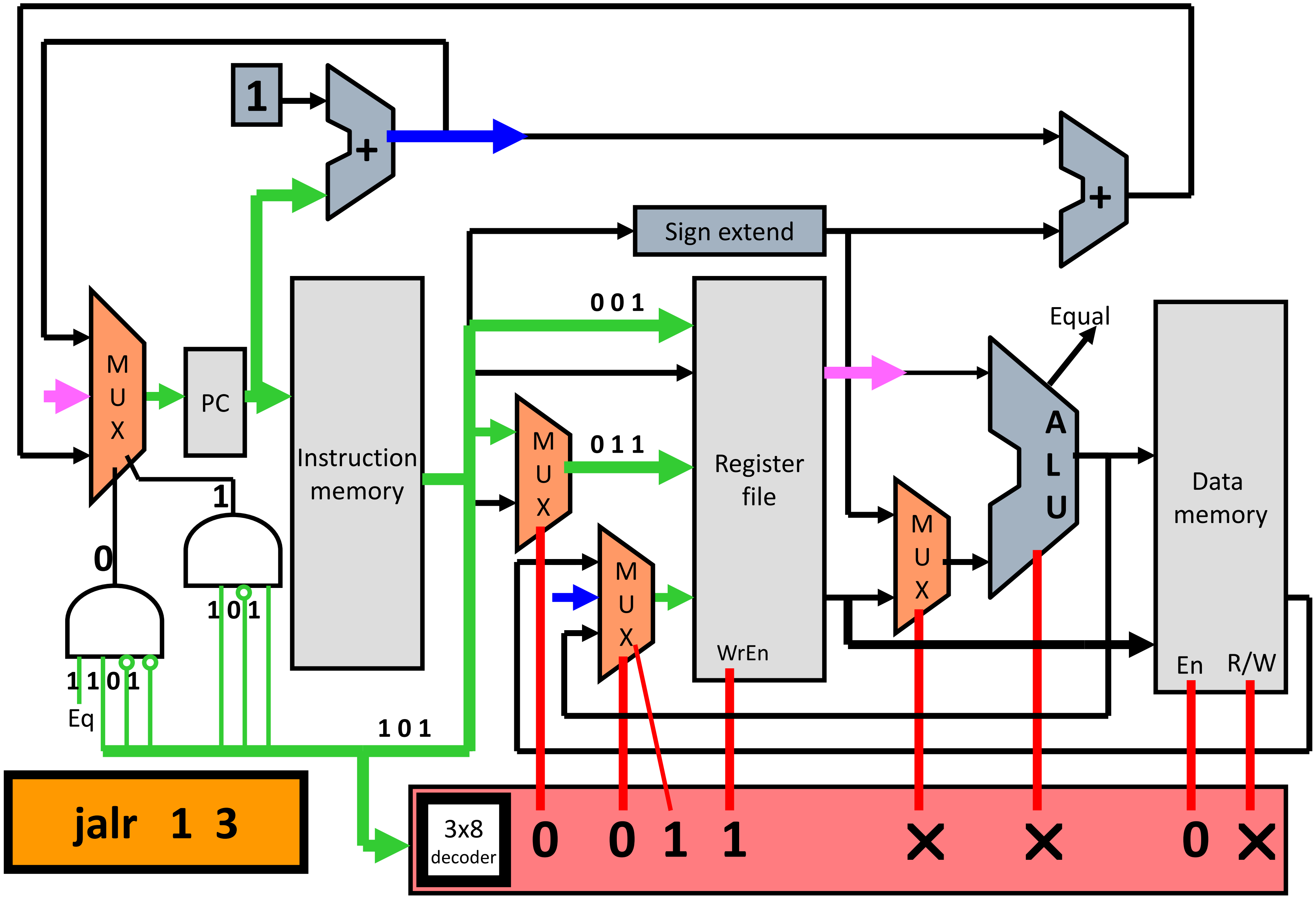
- Every architecture seems to have at least one "ugly" instruction
  - Something that doesn't elegantly fit in with the hardware we've already included

- For LC2K, that ugly instruction is JALR
  - It doesn't fine into our nice clean datapath

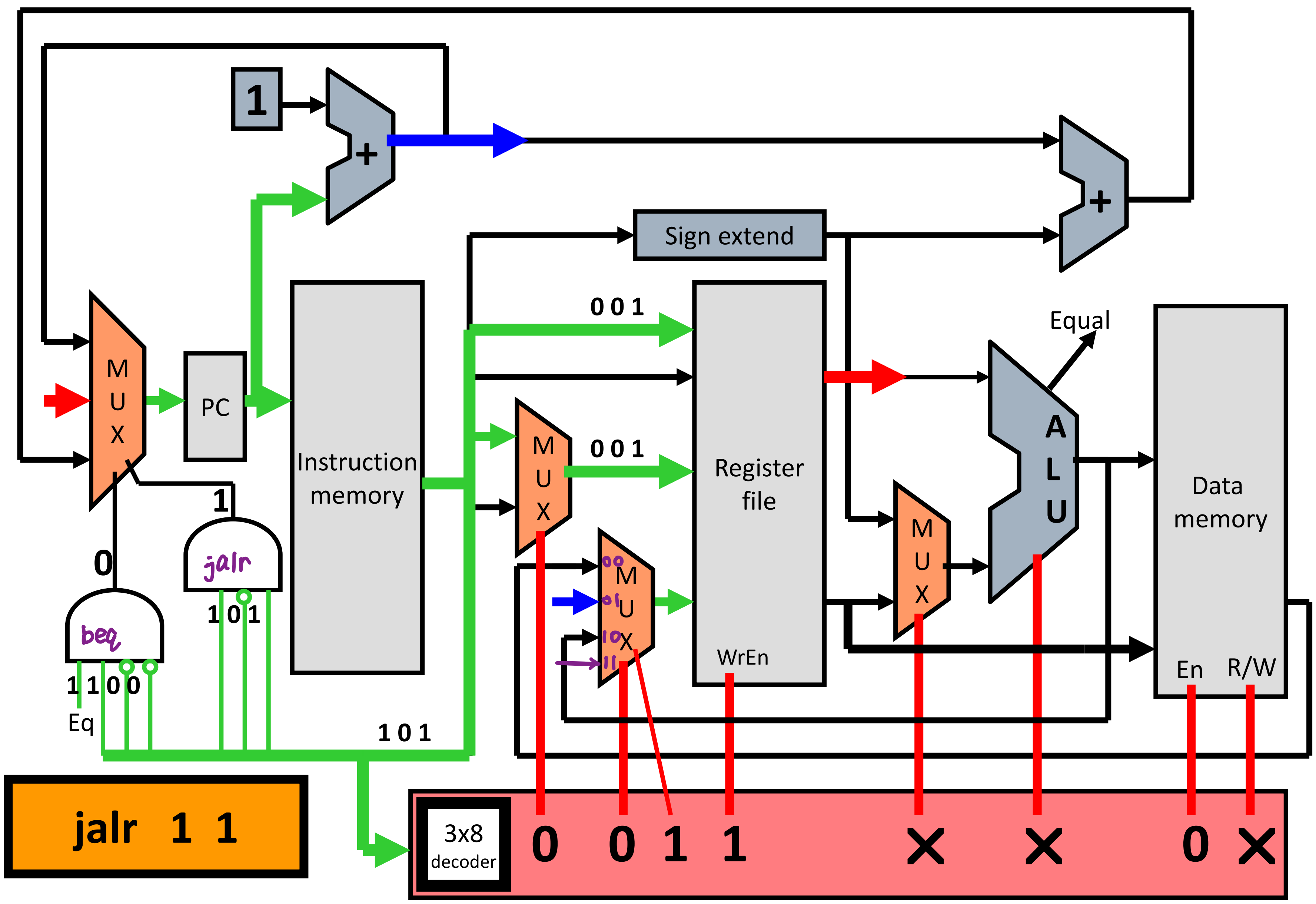
- To implement JALR we need to:
  - Write PC+1 into regB
  - Move regA into PC

- Right now there is:
  - No path to write PC+1 into a register
  - No path to write a register to the PC

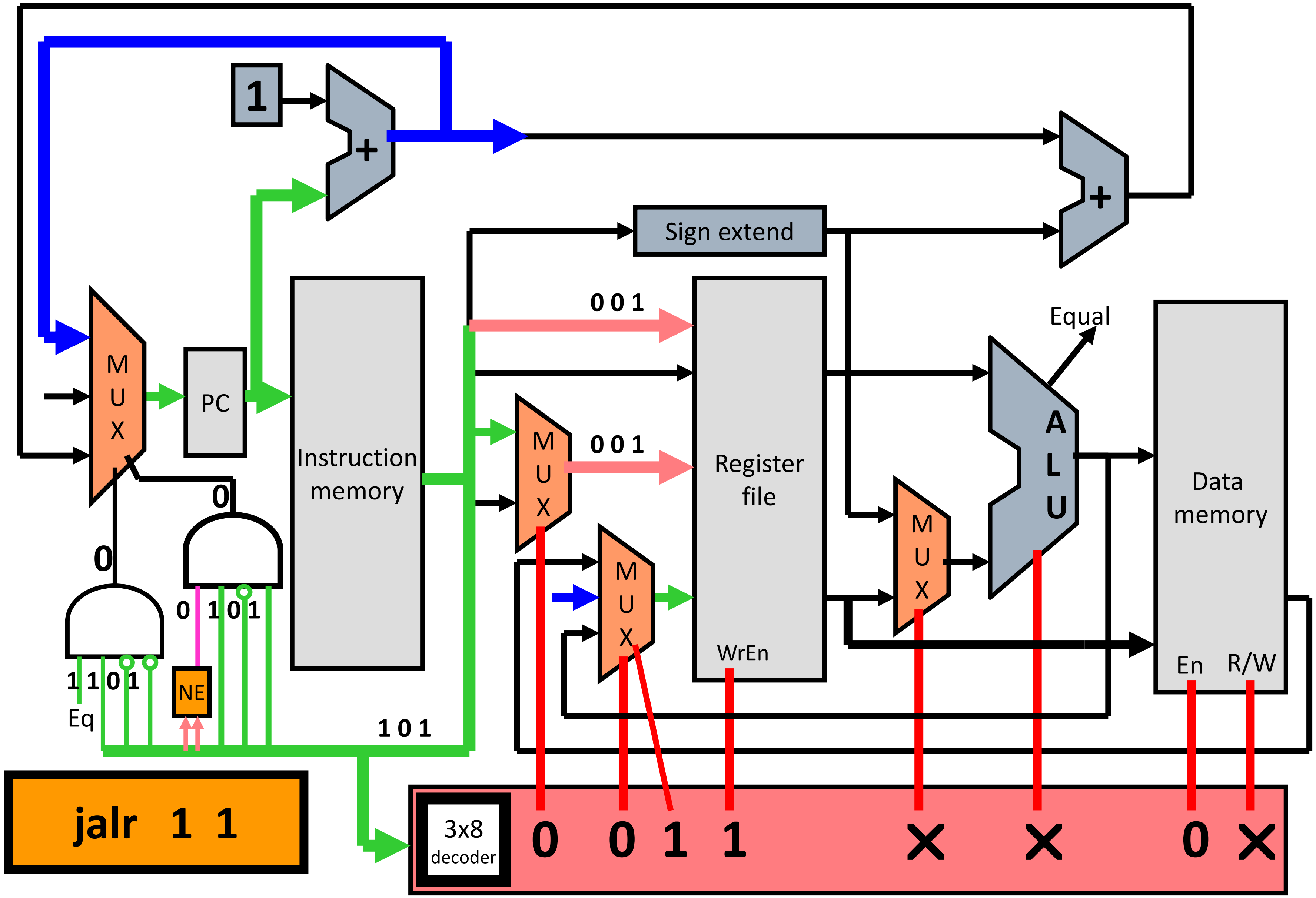
Executing a JALR Instruction



What if regA = regB for JALR?



Changes for JALR 1 1 Instruction



What's Wrong with Single-Cycle?

- **All instructions run at the speed of the slowest instruction.**
- Adding a long instruction can hurt performance
  - What if you wanted to include multiply?
- You cannot reuse any parts of the processor
  - We have 3 different adders to calculate PC+1, PC+1+offset and the ALU
- No benefit in making the common case fast
  - Since every instruction runs at the slowest instruction speed
    - This is particularly important for loads as we will see later

What's Wrong with Single-Cycle?

- 1 ns – Register read/write time
- 2 ns – ALU/adder
- 2 ns – memory access
- 0 ns – MUX, PC access, sign extend, ROM

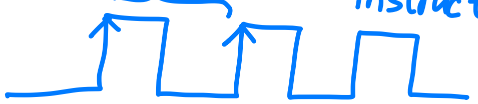
ns = 10^{-9} s

Poll: What is the latency of lw?

	Get Instr	read reg	ALU oper.	mem	write reg	
• add:	2ns	+ 1ns	+ 2ns		+ 1 ns	= 6 ns
• beq:	2ns	+ 1ns	+ 2ns			= 5 ns
• sw:	2ns	+ 1ns	+ 2ns	+ 2ns		= 7 ns
• lw:	2ns	+ 1ns	+ 2ns	+ 2ns	+ 1ns	= 8 ns

The reason we don't count PC=PC+1 (2ns) is because we can do PC=pc+1 while we load instruction from memory.

during one cycle, we need to finish on instruction. depend on the longest one.



the frequency of our clock can't change

Computing Execution Time

Assume: 100 instructions executed

- 25% of instructions are loads, 25 lw 8ns.
- 10% of instructions are stores, 10 SW 7ns,
- 45% of instructions are adds, and 45 add 6ns,
- 20% of instructions are branches. 20 beq 5ns

Single-cycle execution:

?? $100 \times 8ns = 800ns.$

Optimal execution:

?? $25 \times 8 + 10 \times 7 + 45 \times 6 + 20 \times 5.$

$200 + 70 + 270 + 100 = 640ns.$

Poll: **What is the single-cycle execution time?**

How fast could this run if we weren't limited by a single-clock period?

Computing Execution Time

Assume: 100 instructions executed

- 25% of instructions are loads,
- 10% of instructions are stores,
- 45% of instructions are adds, and
- 20% of instructions are branches.

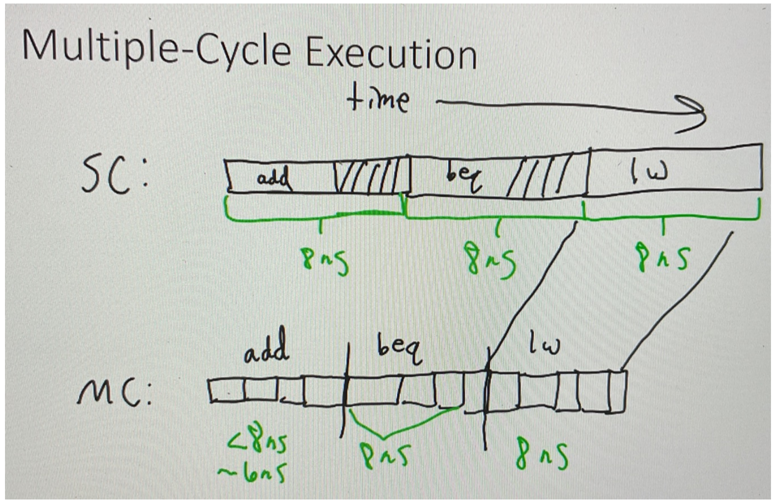
Single-cycle execution:

100 * 8ns = **<u>800</u>** ns

Optimal execution:

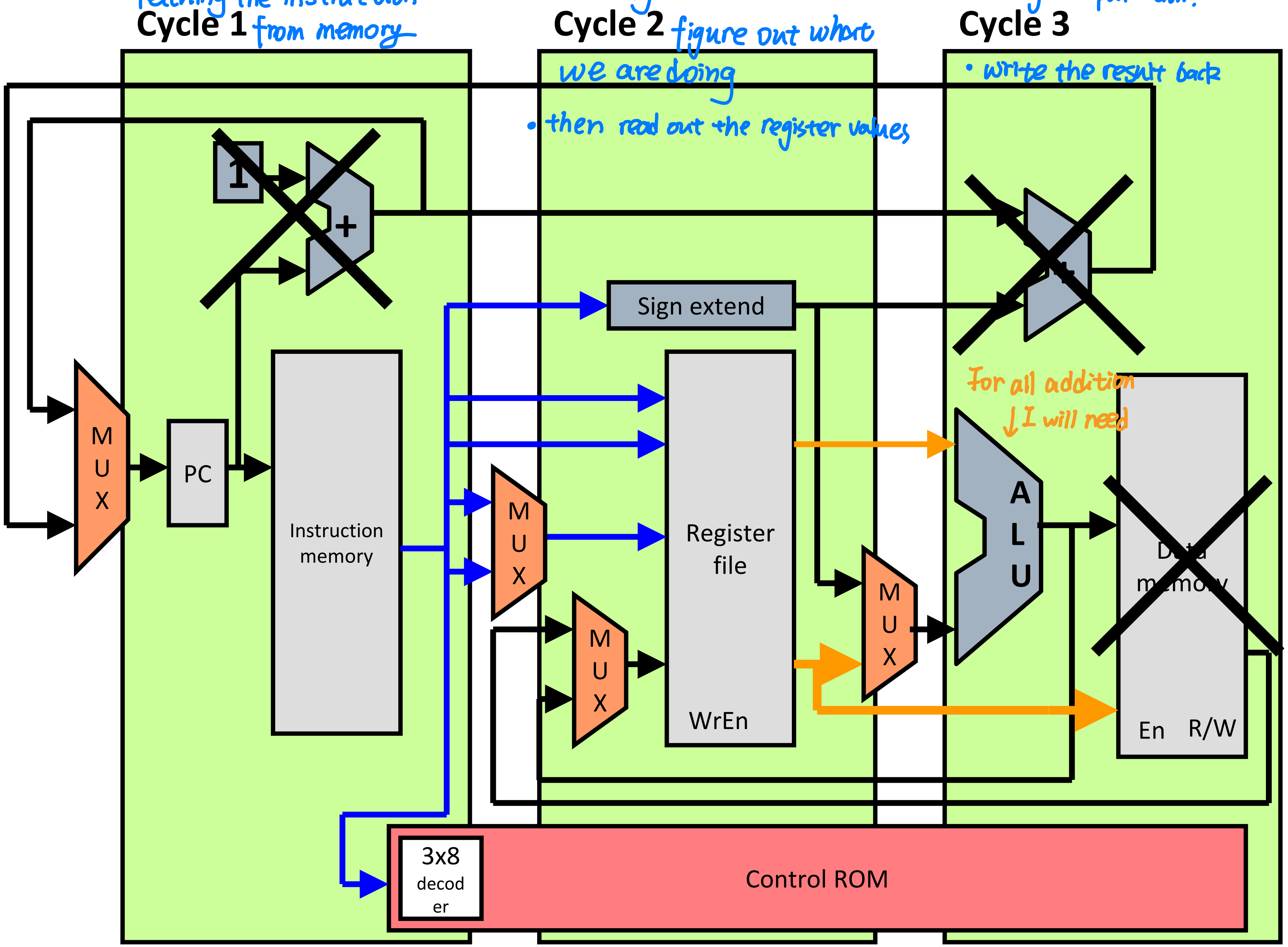
25*8ns + 10*7ns + 45*6ns + 20*5ns = **<u>640</u>** ns

Multiple-Cycle Execution



- Each instruction takes multiple cycles to execute
  - Cycle time is reduced
  - Slower instructions take more cycles
  - Faster instruction take fewer cycles
    - We can start next instruction earlier, rather than just waiting
  - Can reuse datapath elements each cycle
- What is needed to make this work?
  - Since you are re-using elements for different purposes, you need more and/or wider MUXes.
  - You may need extra registers if you need to remember an output for 1 or more cycles.
  - Control is more complicated since you need to send new signals on each cycle.

LC2K Datapath – cycle groups



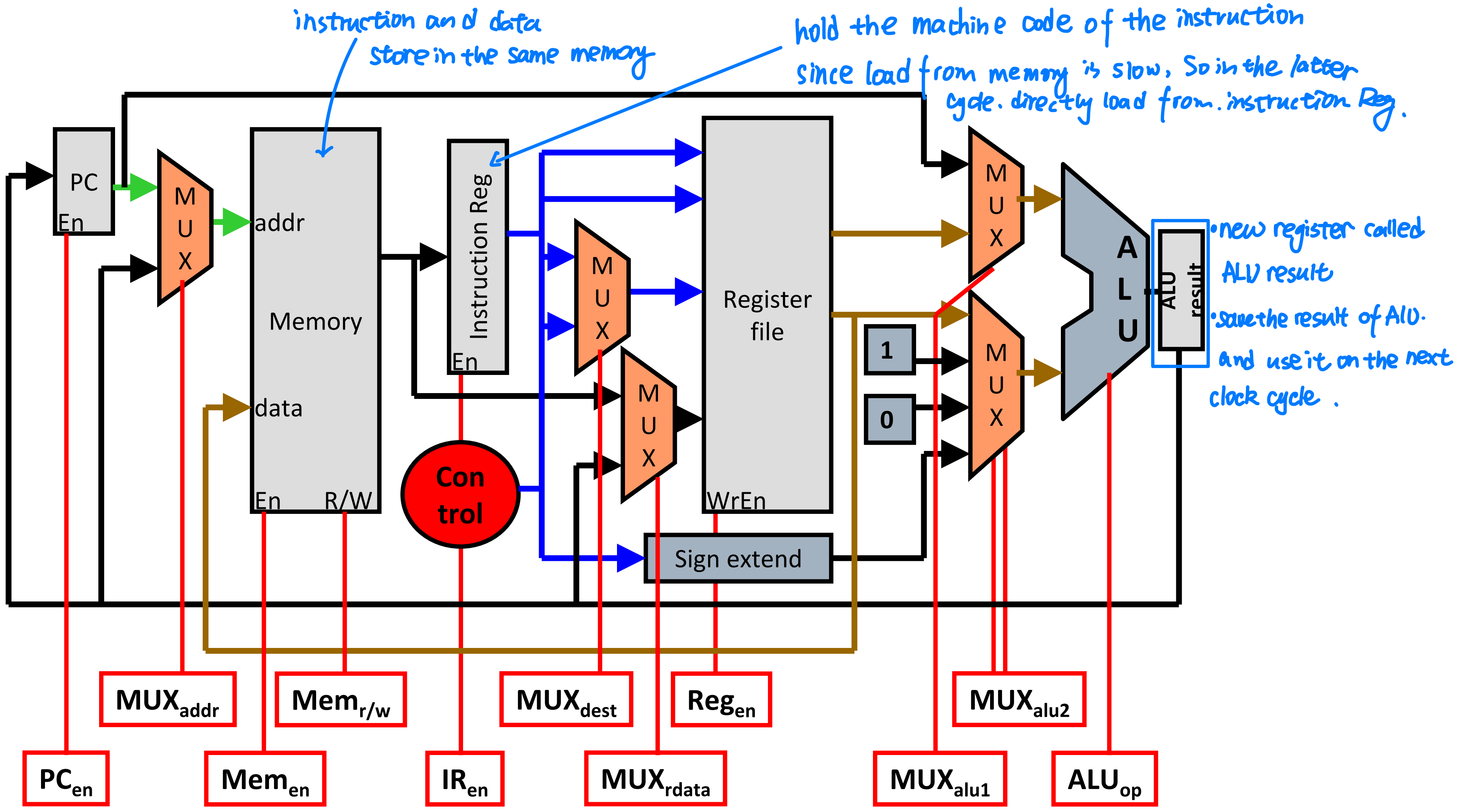
Fetching the instruction from memory

- Decoding the instruction figure out what we are doing
- then read out the register values

- excute the instruction.
- doing computation.
- write the result back

For all addition ↓ I will need

Multi-cycle LC2 Datapath



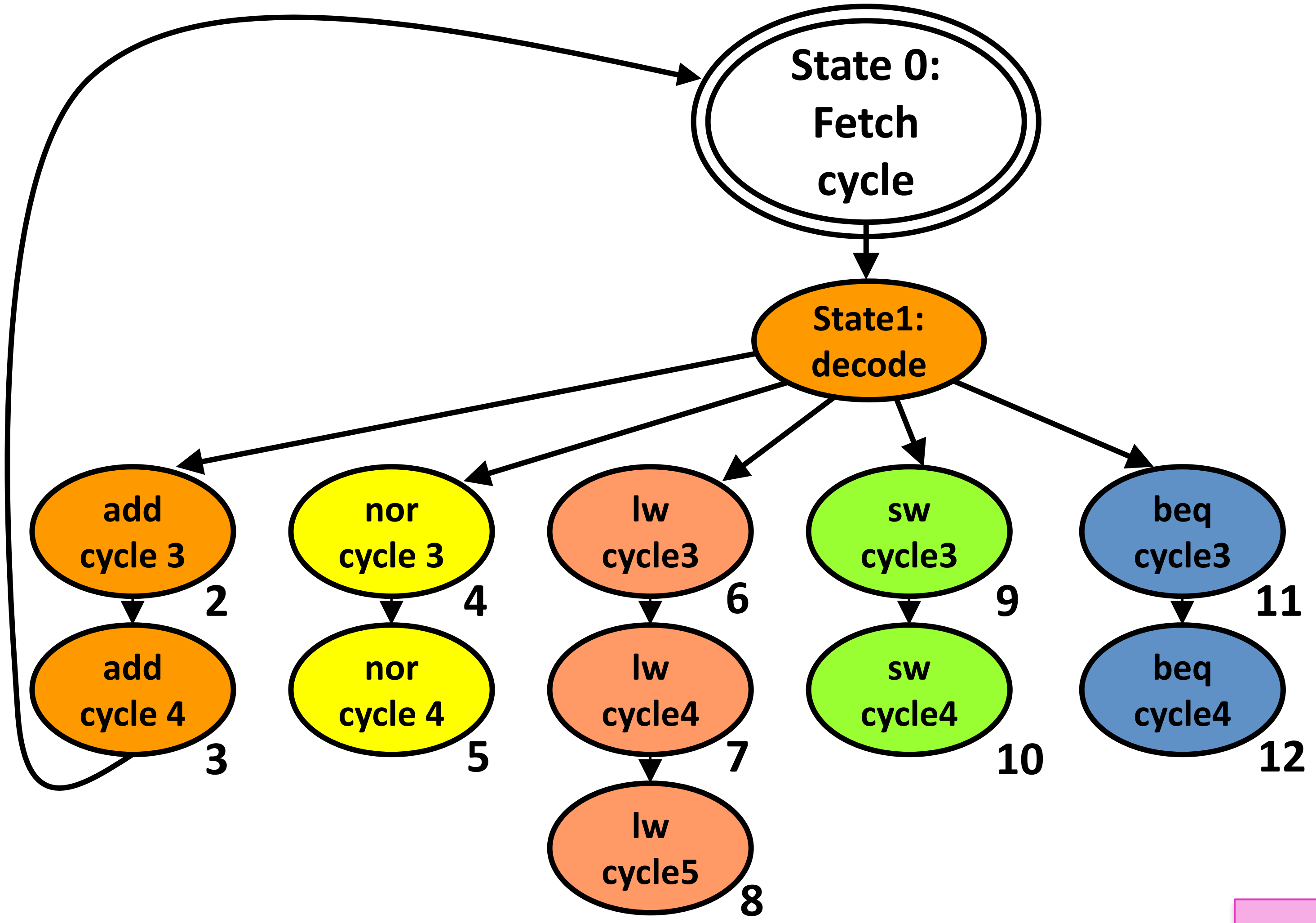
instruction and data store in the same memory

hold the machine code of the instruction since load from memory is slow, So in the latter cycle. directly load from. instruction Reg.

- new register called ALU result
- save the result of ALU. and use it on the next clock cycle.

Each red signal comes from "Control" (implemented via ROM as before)

State machine for multi-cycle control signals (transition functions)



Note: we aren't worrying about JALR instruction in hardware going forward

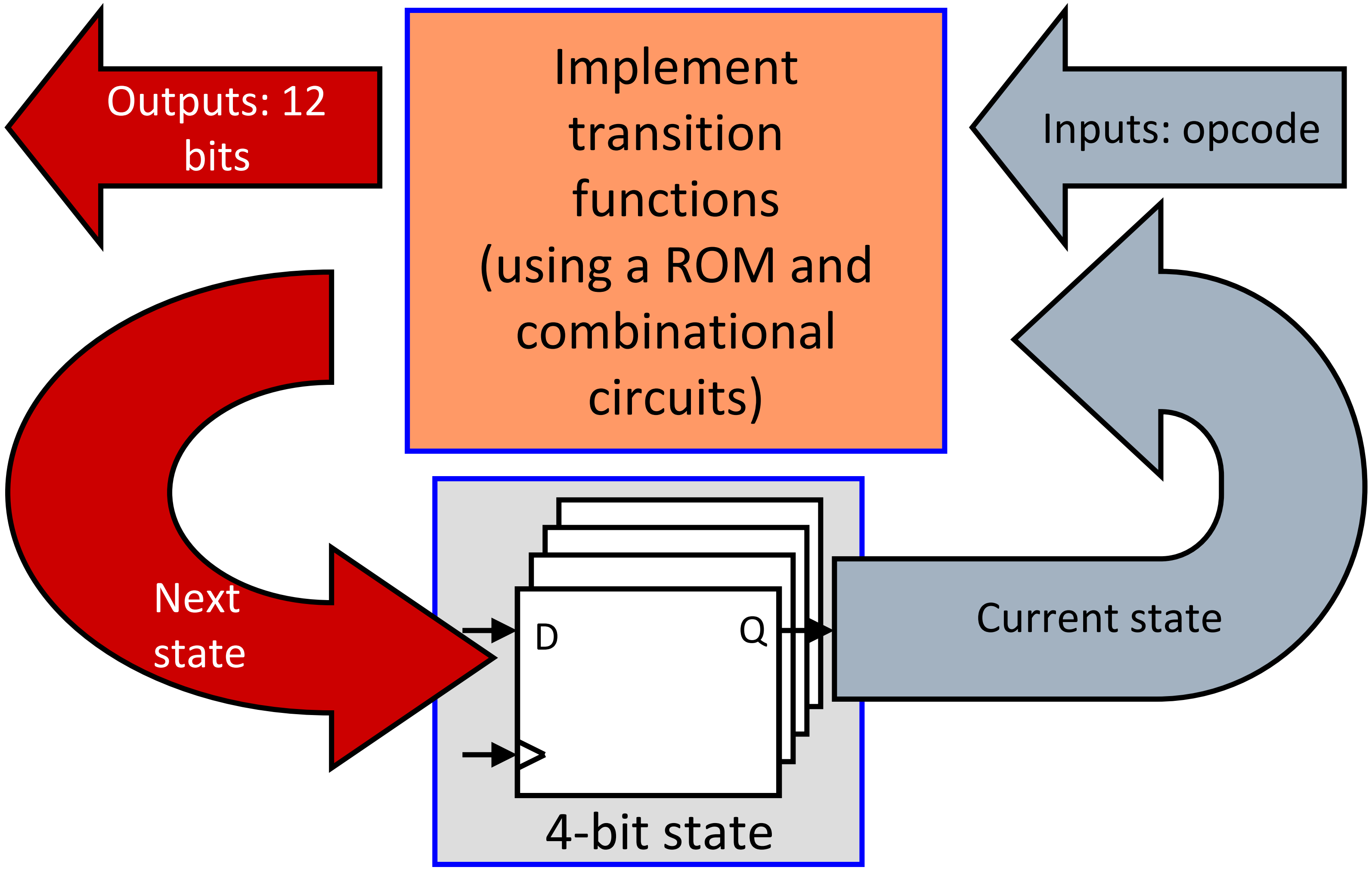
Poll: How many bits of storage are needed to store the state?

13 states

All instruction need go back to state 0.

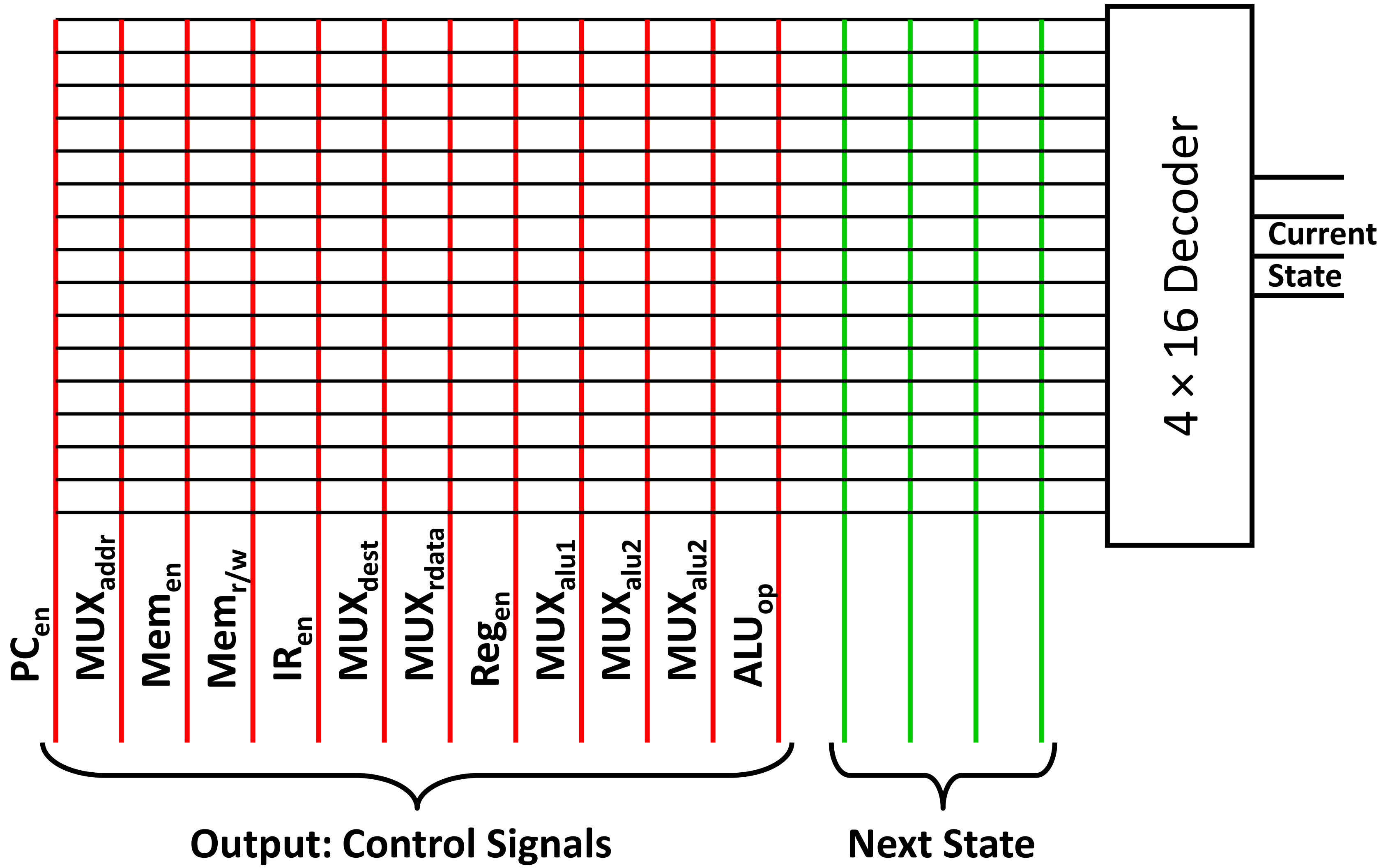
4bits → 4 ff.

Implementing FSM



Outputs: 12 bits
Implement transition functions (using a ROM and combinational circuits)
Inputs: opcode
Next state
D
Q
Current state
4-bit state

Building the Control ROM

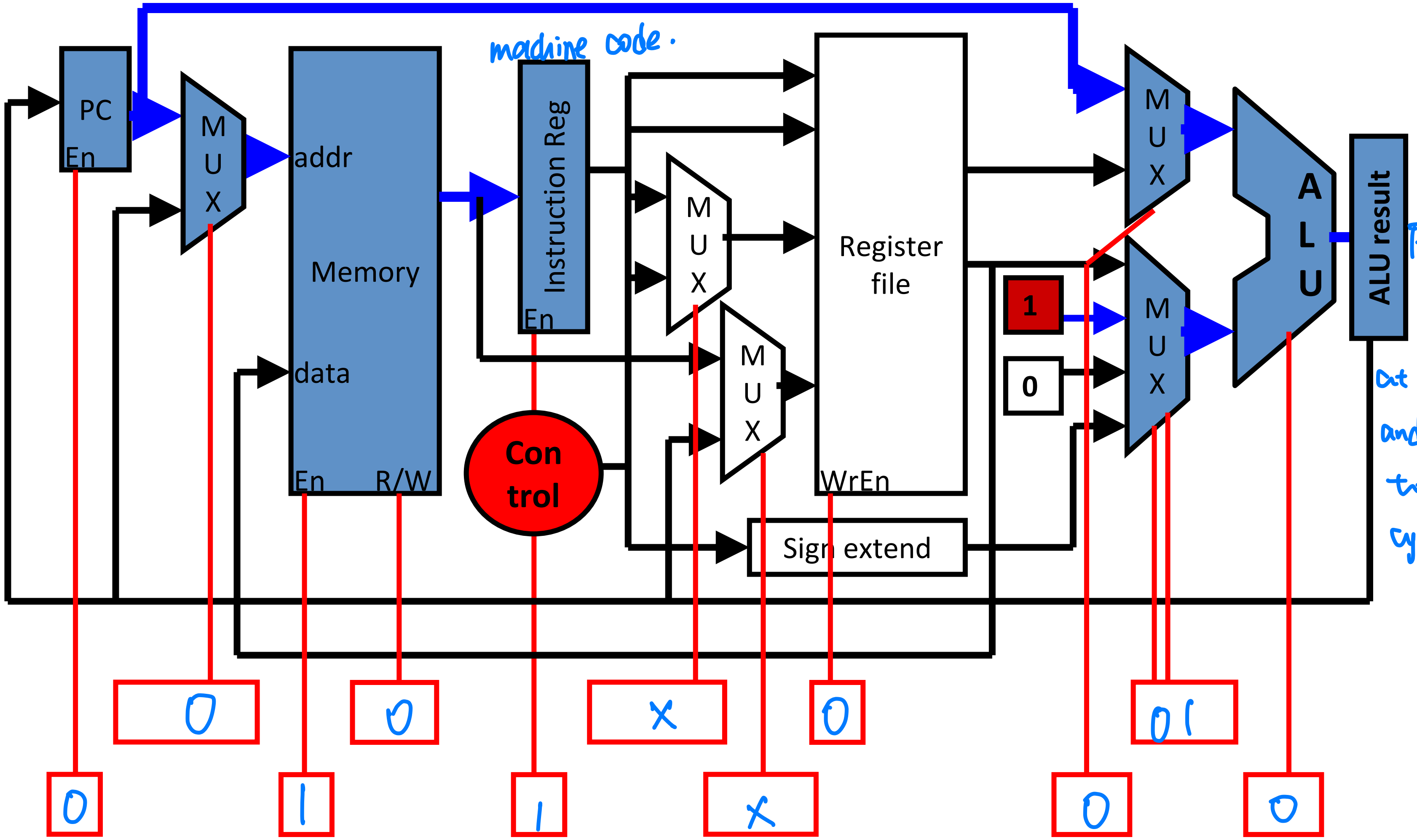


First Cycle (State 0) Fetch Instr

- What operations need to be done in the first cycle of executing any instruction?
  - Read memory[PC] and store into instruction register.
    - Must select PC in memory address MUX ($MUX_{addr} = 0$)
    - Enable memory operation ($Mem_{en} = 1$)
    - R/W should be (read) ($Mem_{r/w} = 0$)
    - Enable Instruction Register write ($IR_{en} = 1$)
  - Calculate PC + 1
    - Send PC to ALU ($MUX_{alu1} = 0$)
    - Send 1 to ALU ($MUX_{alu2} = 01$)
    - Select ALU add operation ($ALU_{op} = 0$)
  - $PC_{en} = 0$; $Reg_{en} = 0$; MUX_{dest} and $MUX_{rdata} = X$
- Next State: Decode Instruction

First Cycle (State 0) Fetch Instr

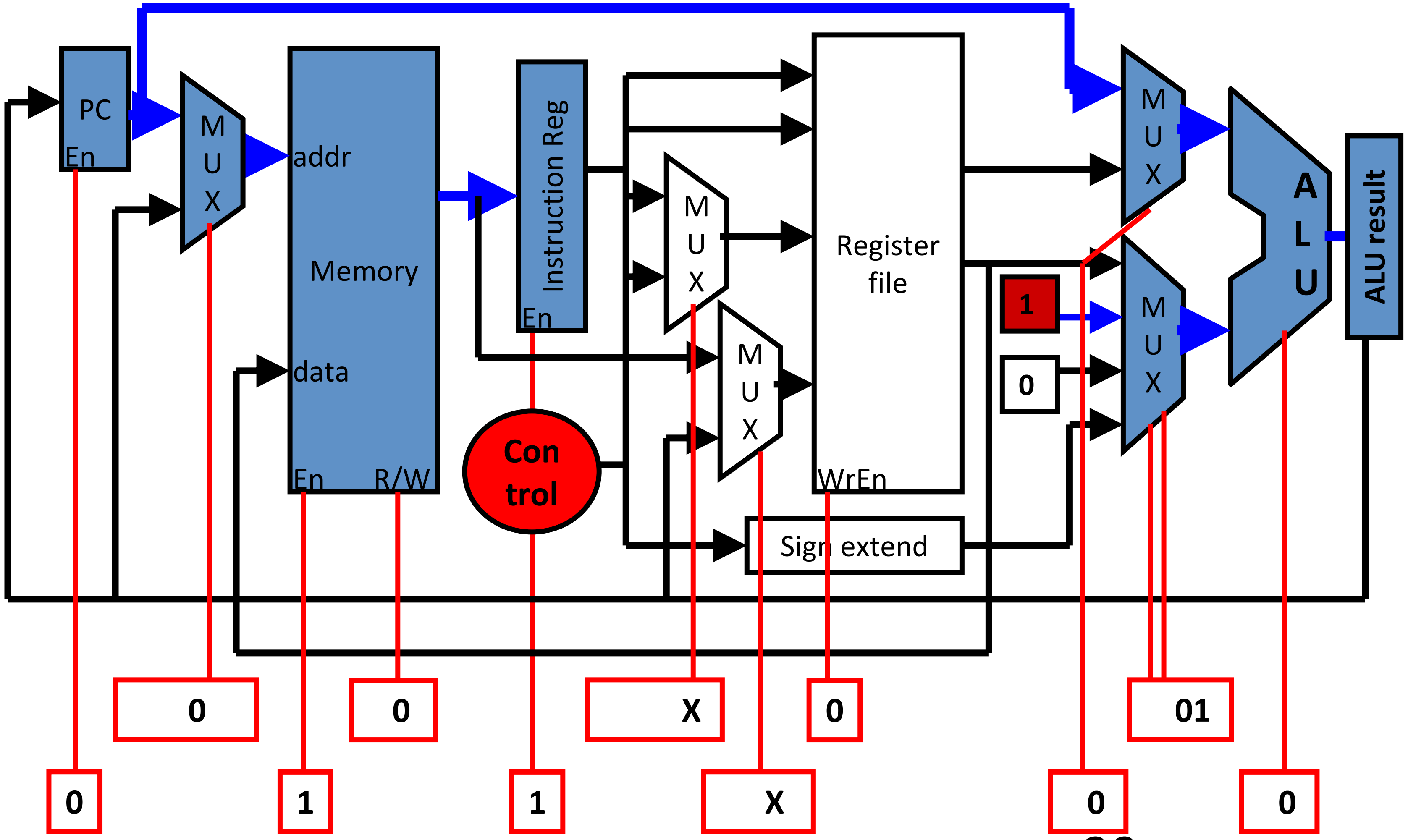
This is the same for all instructions
(since we don't know the instruction yet!)



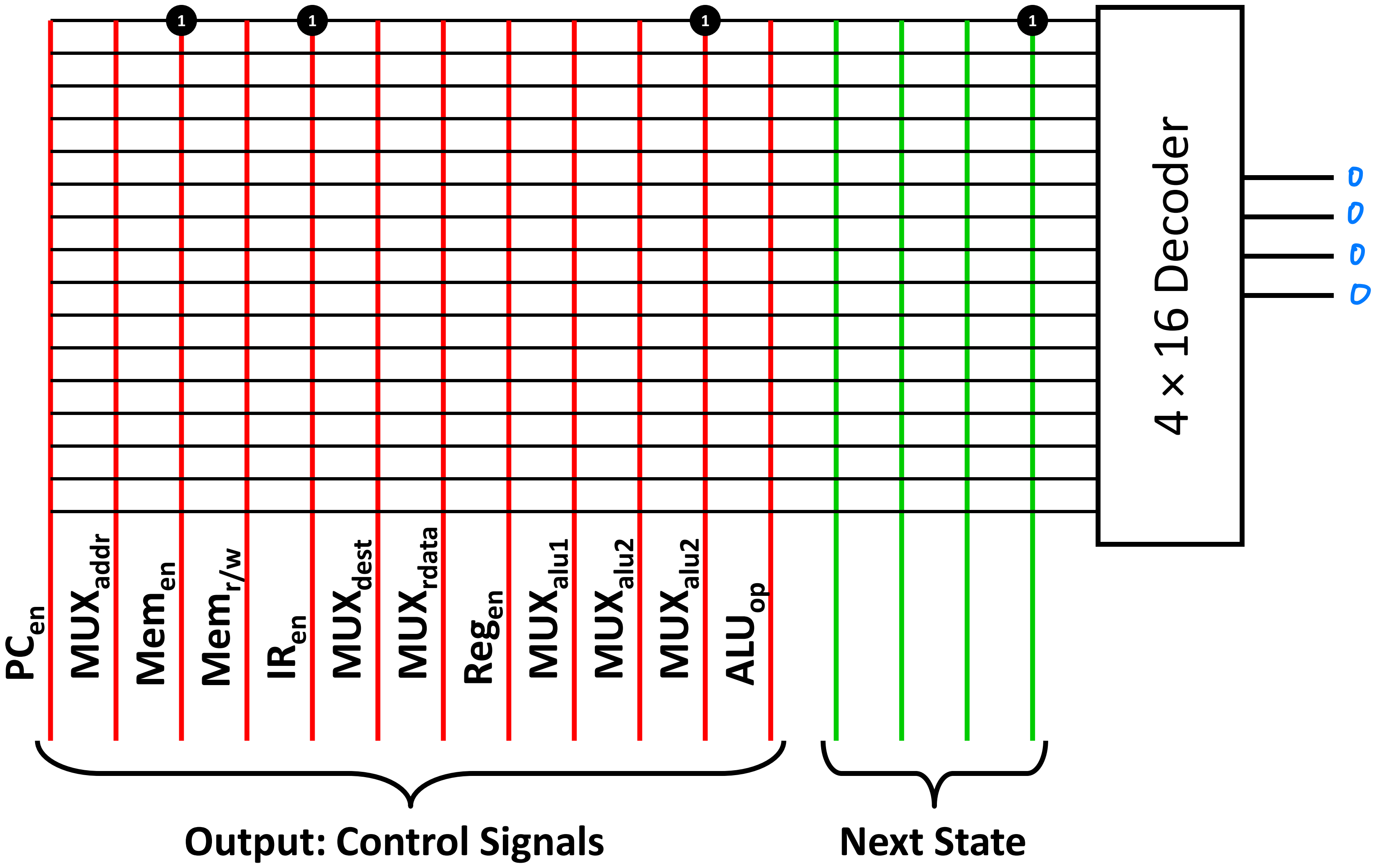
PC+1.
↙
store in ALU reg at the end of this cycle, and will write it back to PC on the next clock cycle.

First Cycle (State 0) Fetch Instr

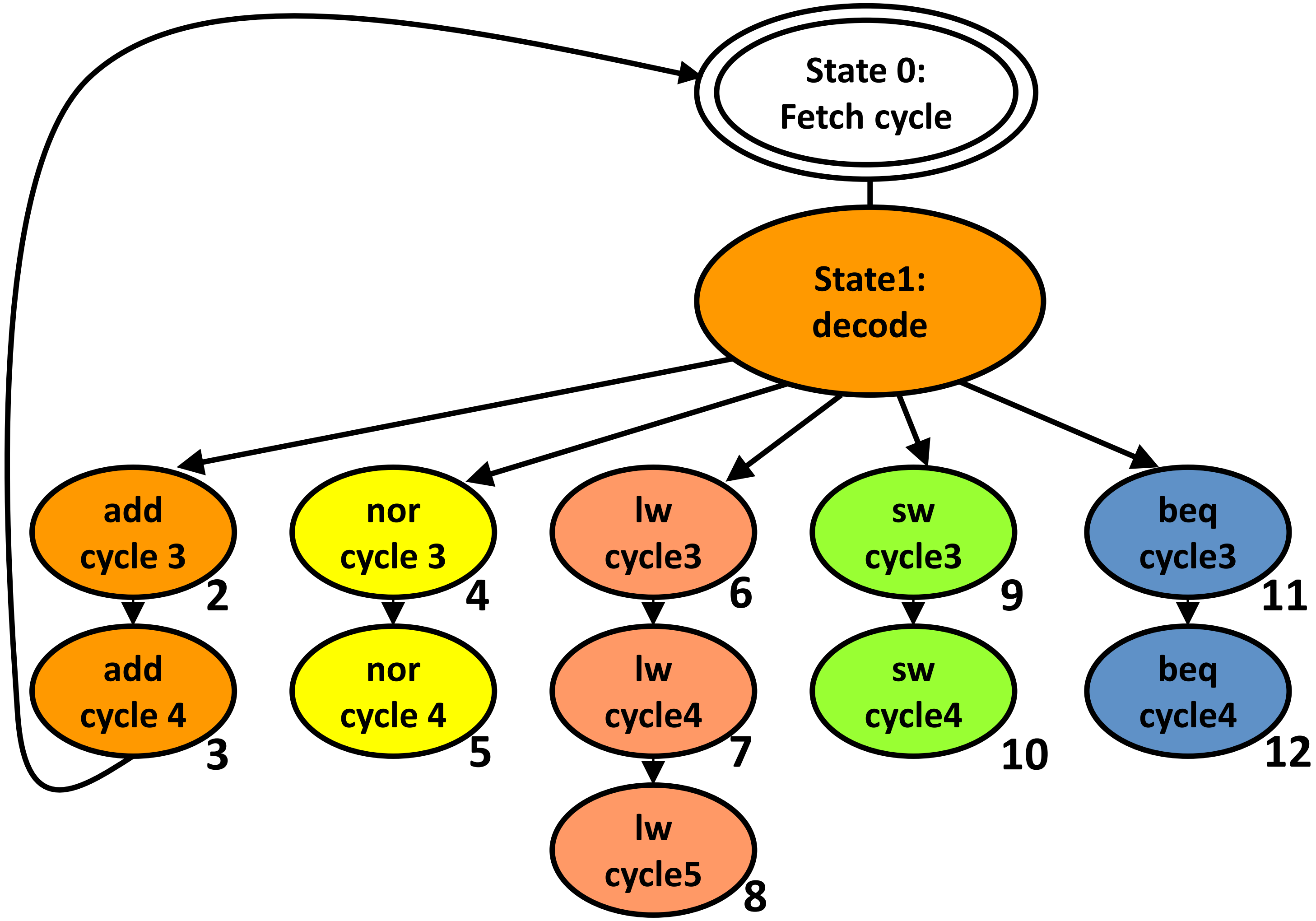
This is the same for all instructions (since we don't know the instruction yet!)



Building the Control ROM



State 1: instruction decode

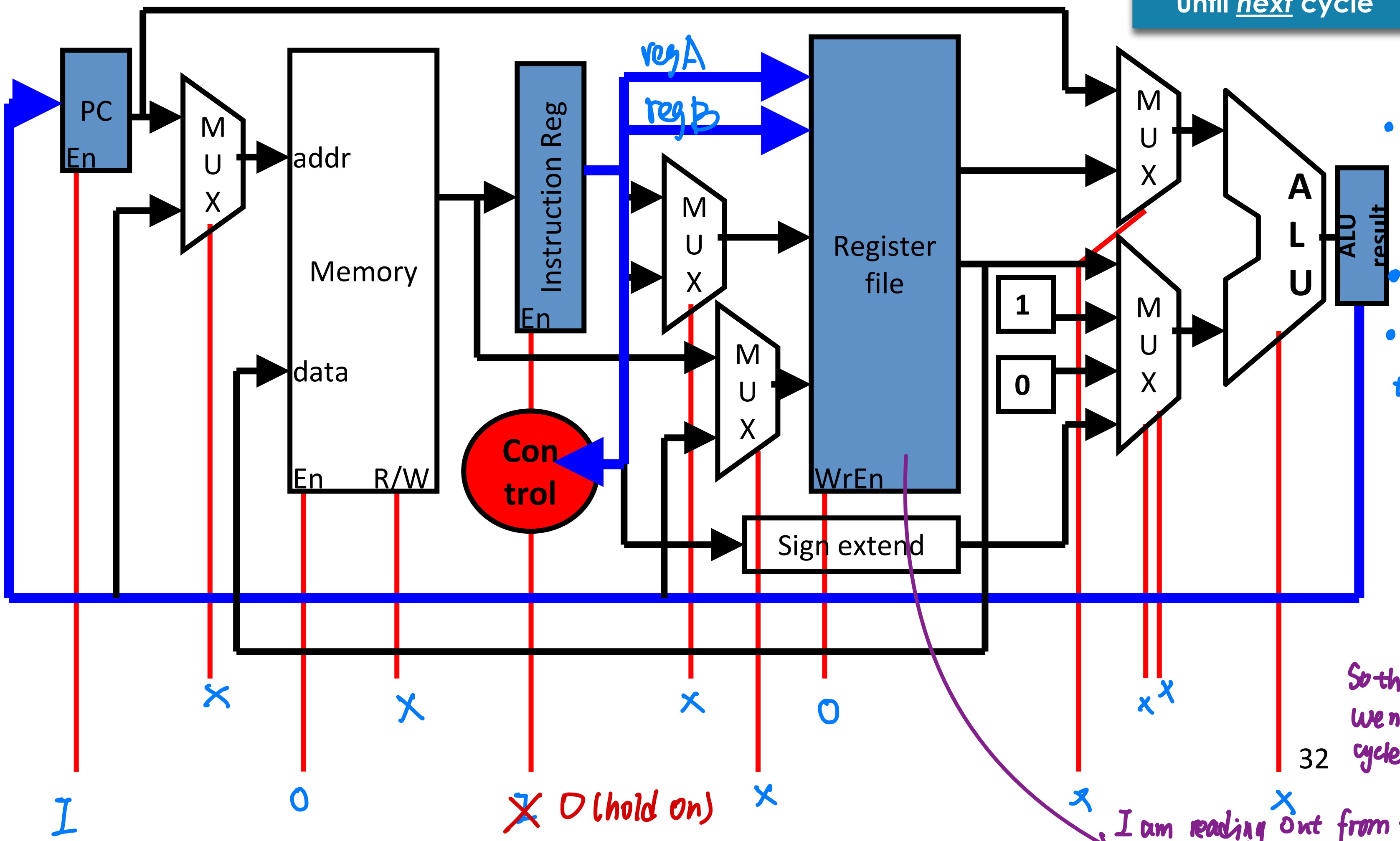


State 1: output function

Poll: What will the control bits be?

Update PC; read registers (regA and regB); use opcode to determine next state

Note: since RF read latency is same as clock period, RF data isn't available until *next* cycle



- In this cycle, send pc+1 value back to updat pc
- read regA regB the
- use the opcode to determine the next state.

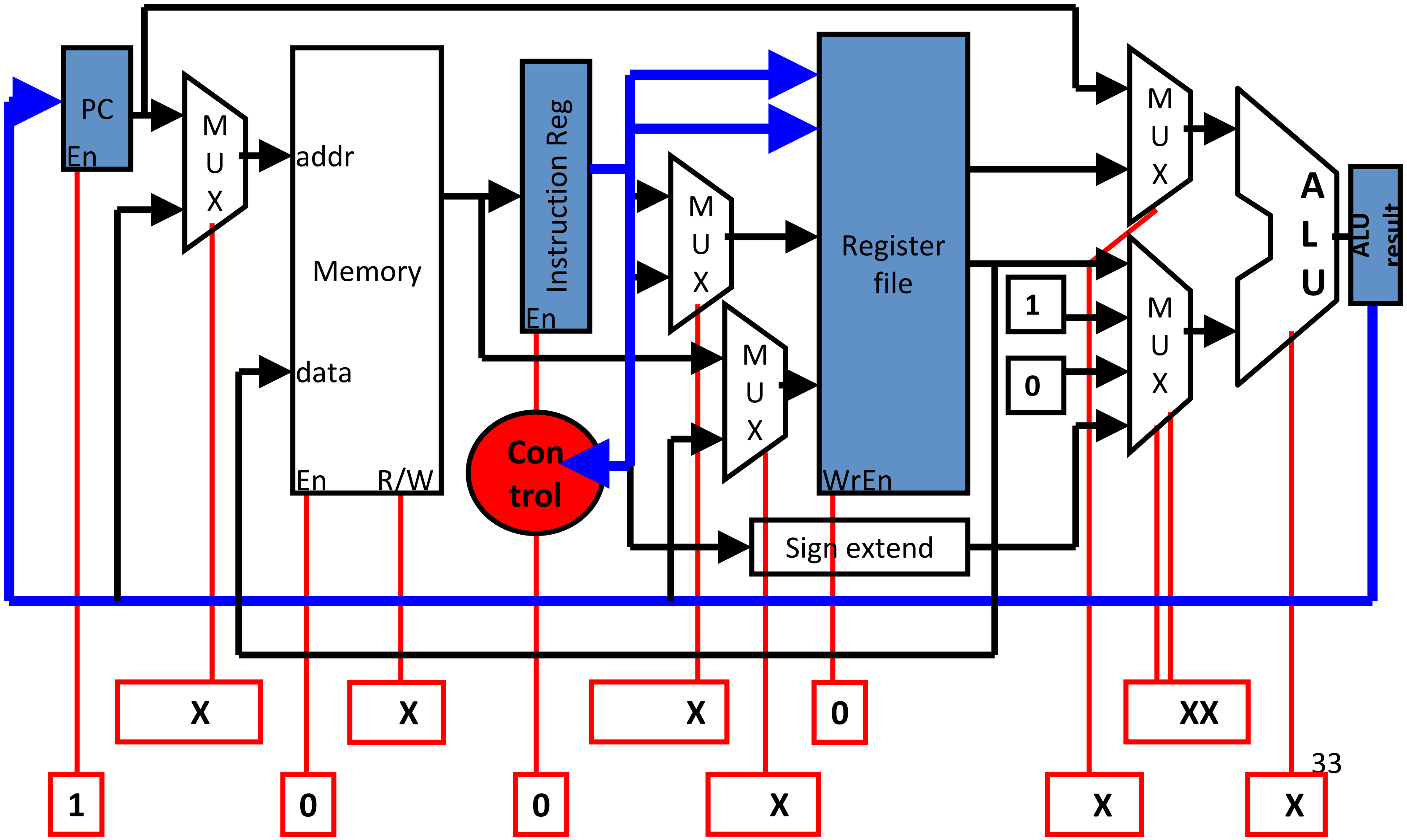
So the value isn't available yet. We need to wait until the next cycle.

We already save the value in the previous cycle and I want to hold on to that.

I am reading out from the register file. But we assume that it will take pretty much the whole clock period to read out the register file

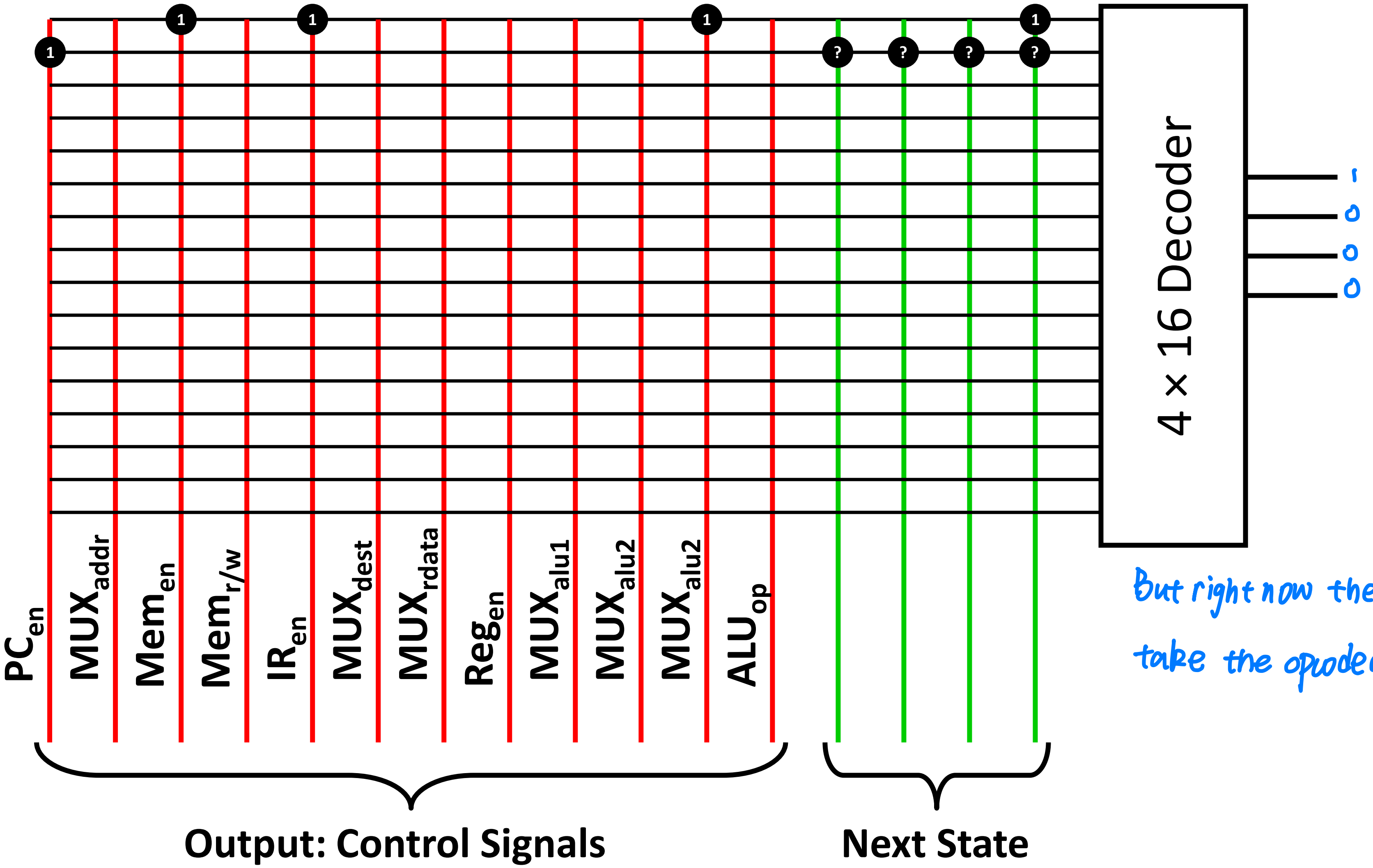
State 1: output function

Update PC; read registers (regA and regB);
use opcode to determine next state



Building the Control ROM

How do we set next state?? Let's come back to that…



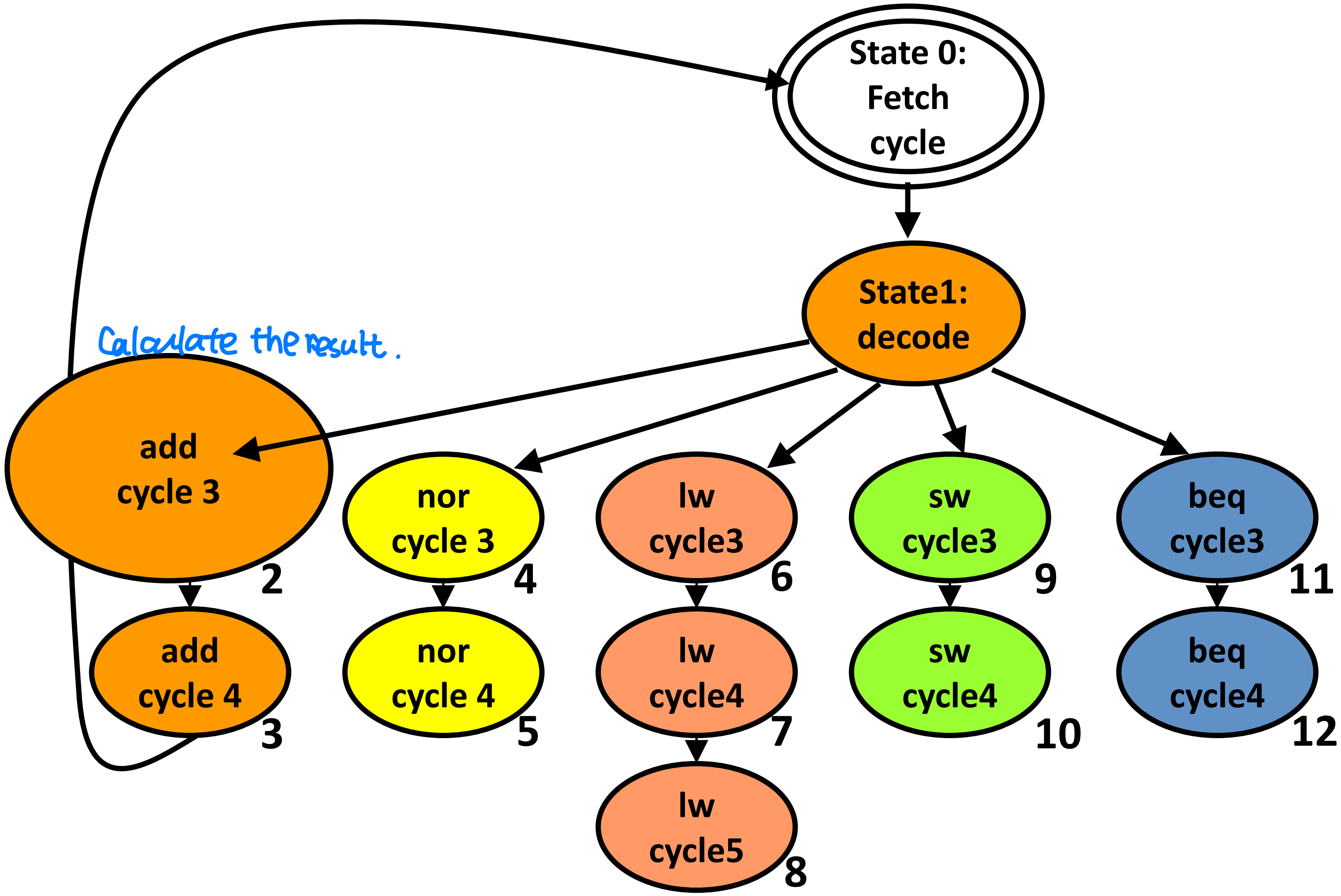
But right now the Decoder doesn't take the opcode as an input.

Next time

- Finish up multi-cycle processors
- Introduce pipelining

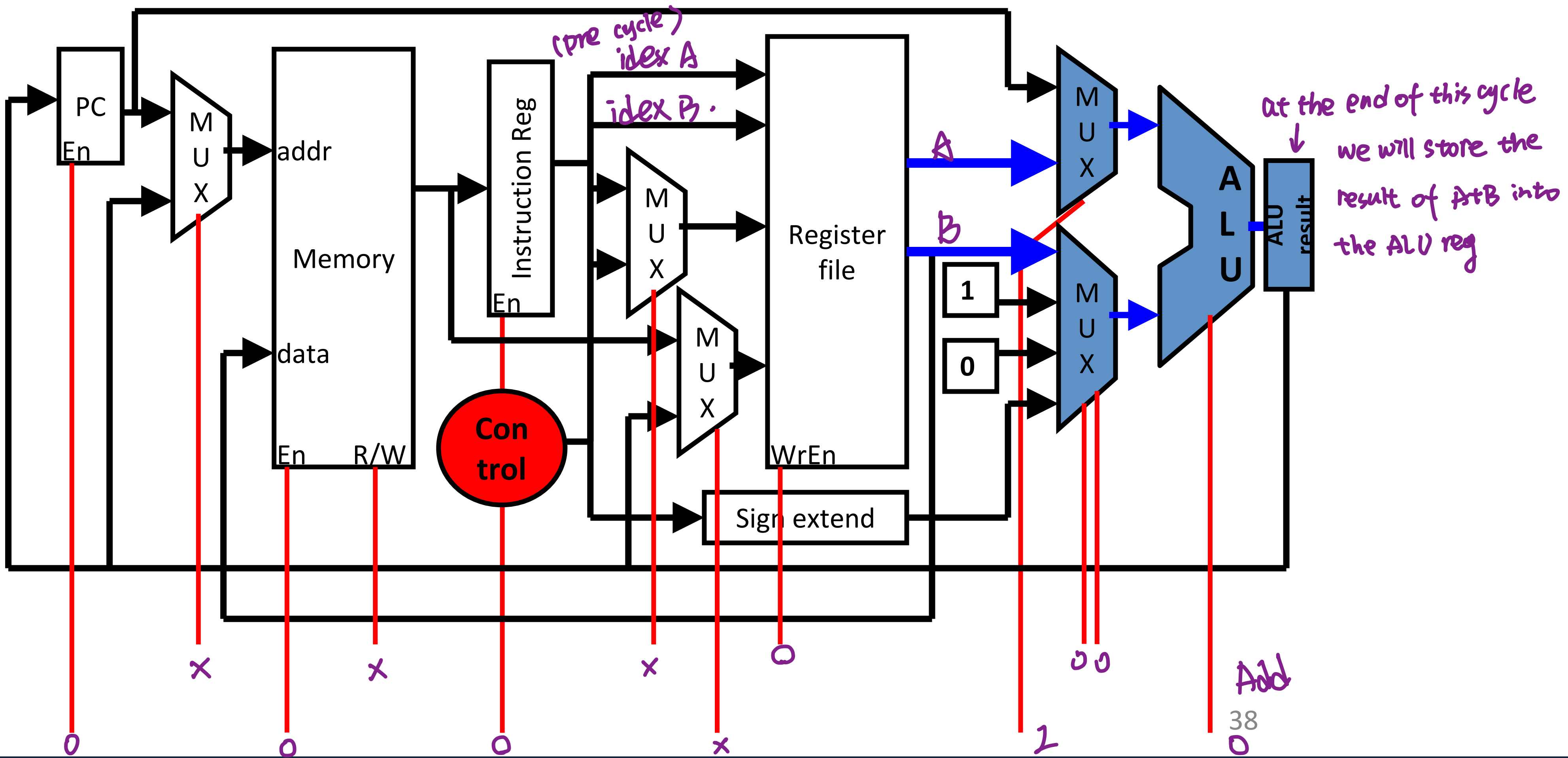
Extra Slides

State 2: Add cycle 3



State 2: Add Cycle 3 Operation

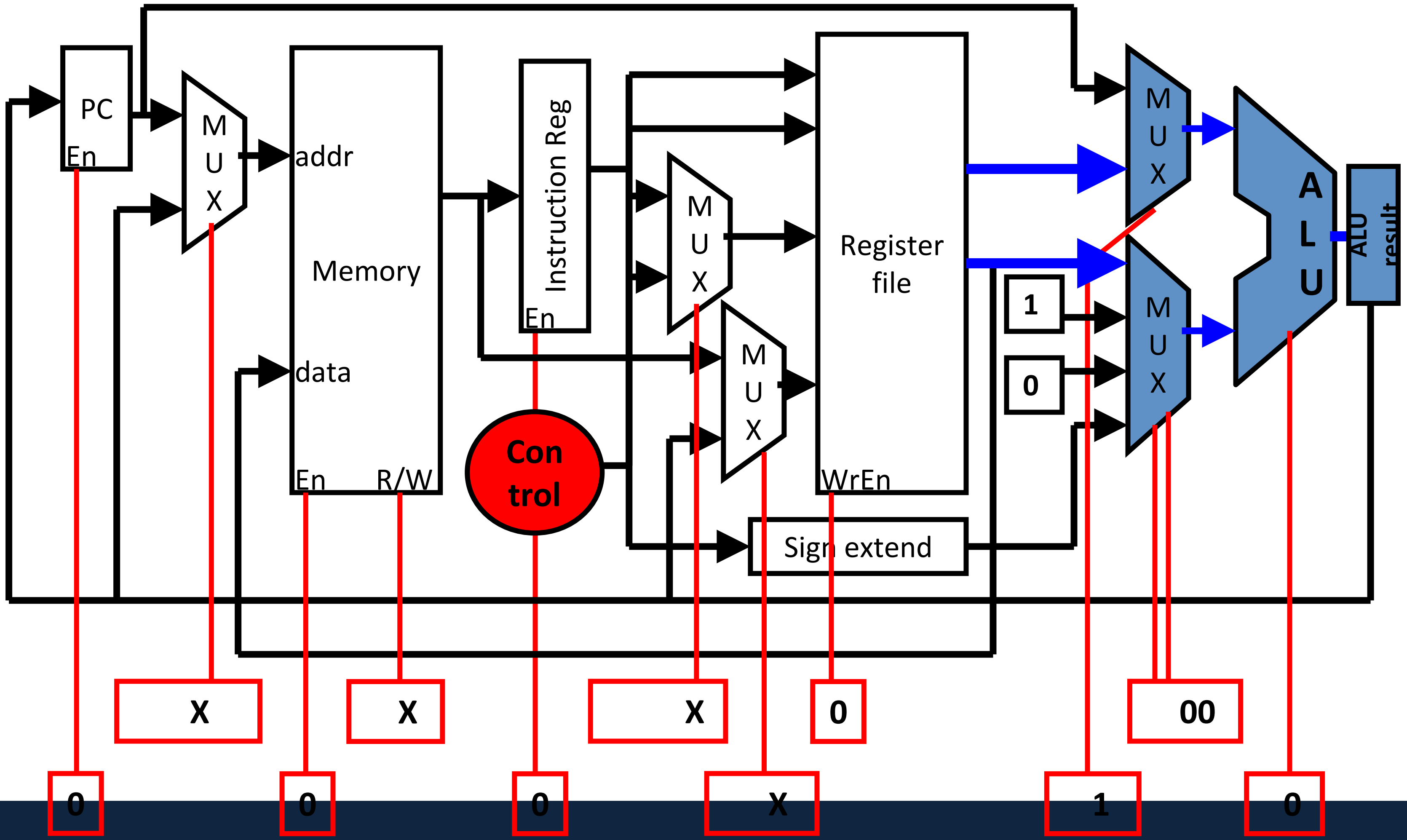
Send control signals to MUX to select values of regA and regB and control signal to ALU to add



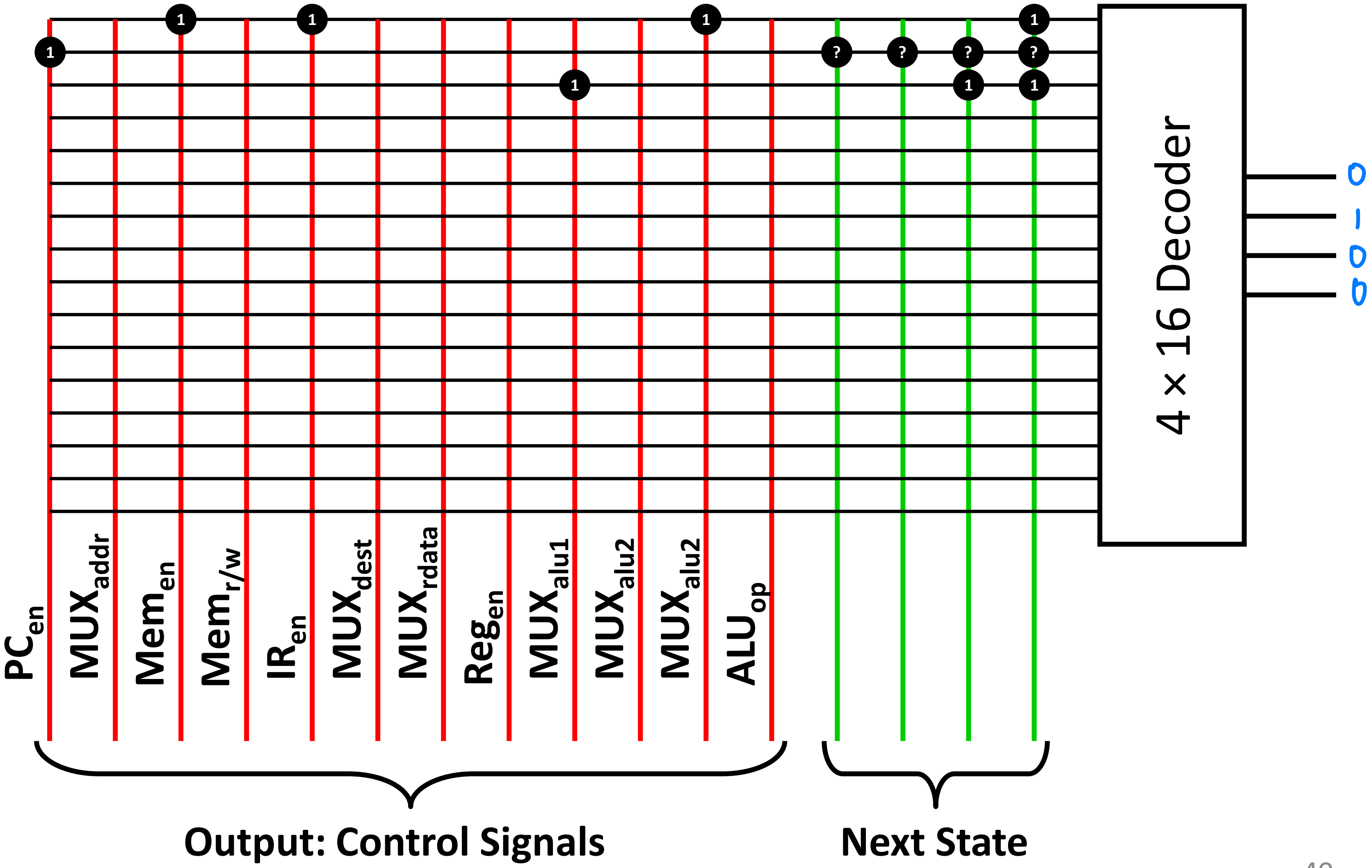
at the end of this cycle ↓ we will store the result of A+B into the ALU reg

State 2: Add Cycle 3 Operation

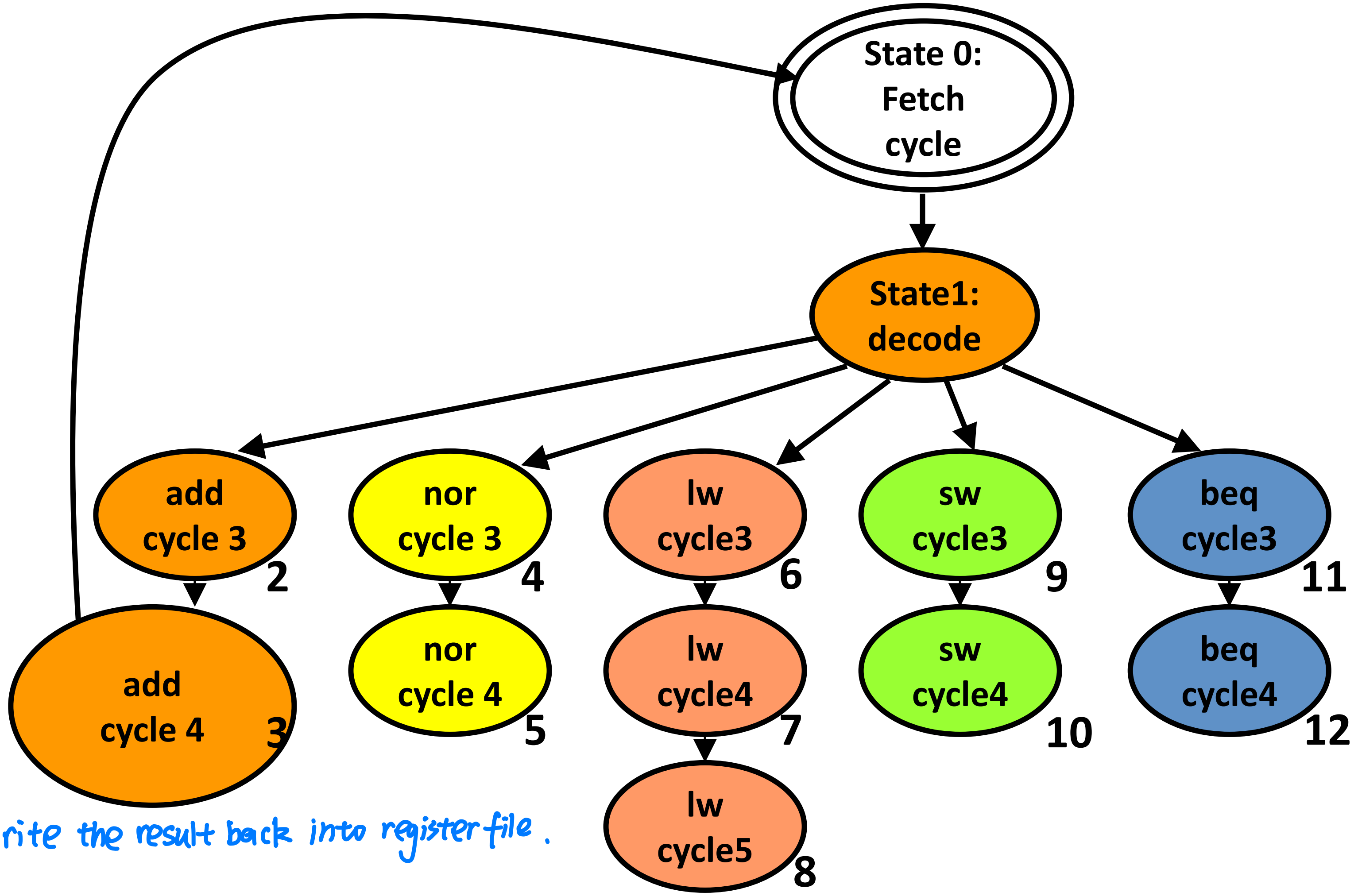
Send control signals to MUX to select values of regA and regB and control signal to ALU to add



Building the Control Rom



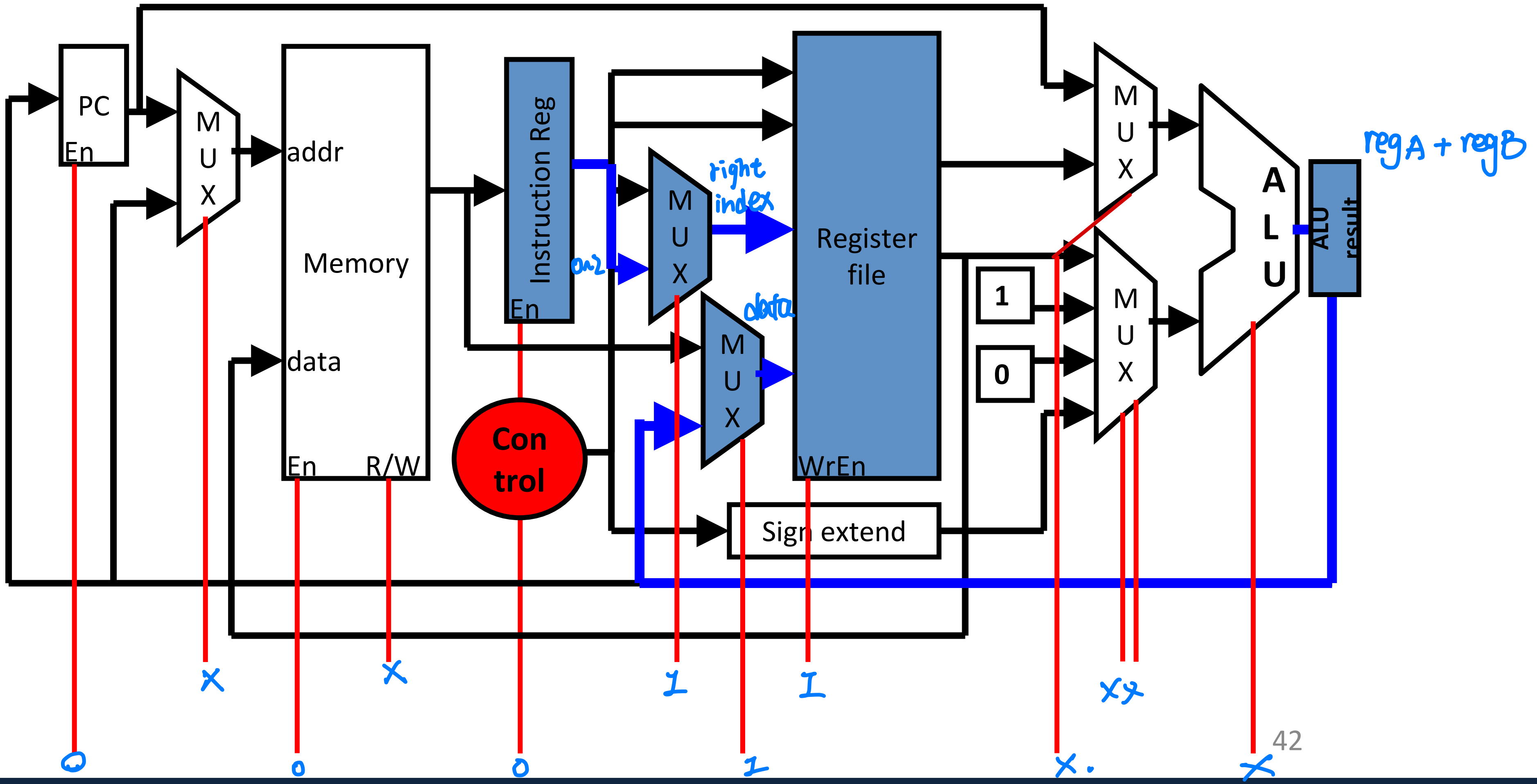
State 3: Add cycle 4



write the result back into register file.

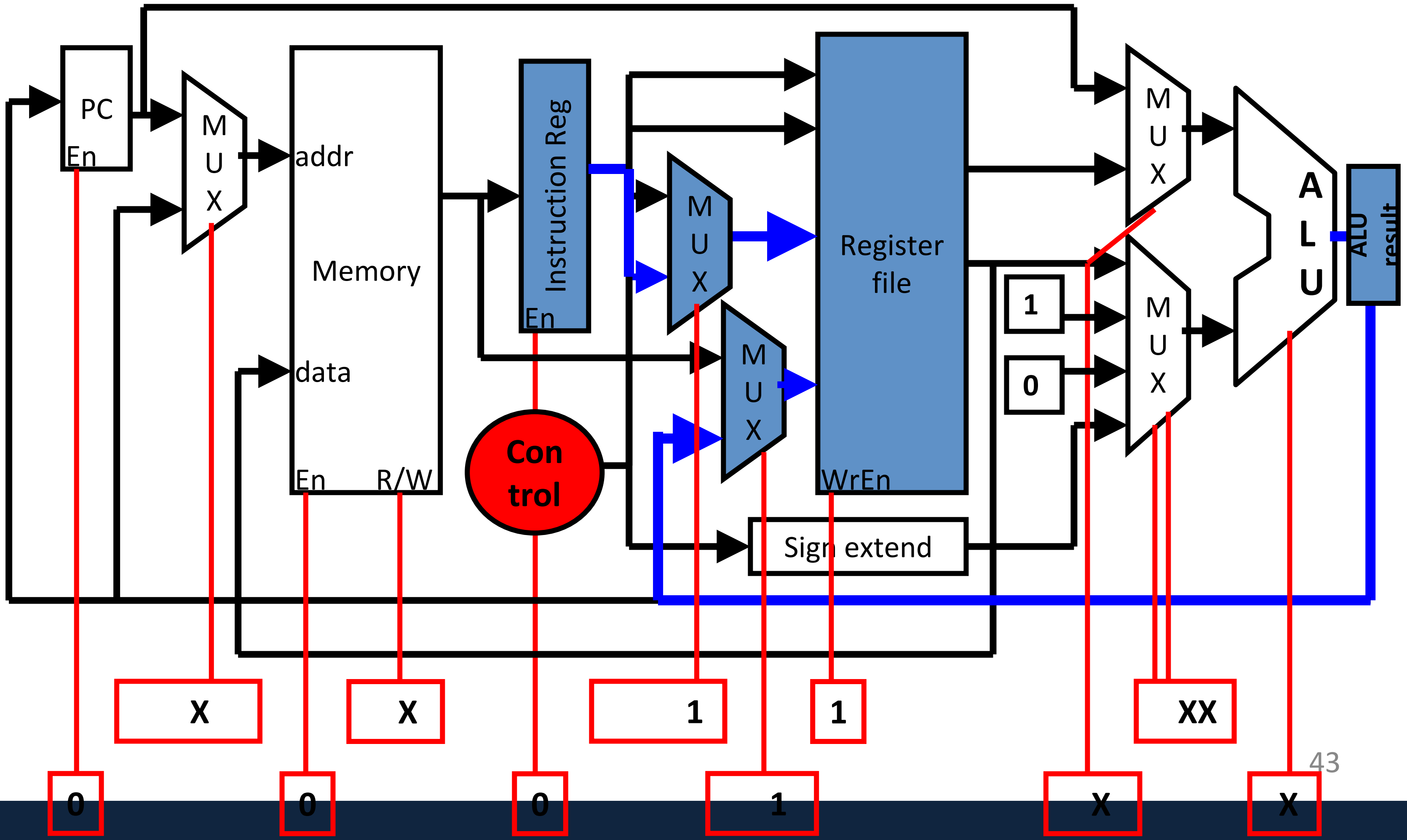
Add Cycle 4 (State 3) Operation

Send control signal to address MUX to select dest and to data MUX to select ALU output, then send write enable to register file.

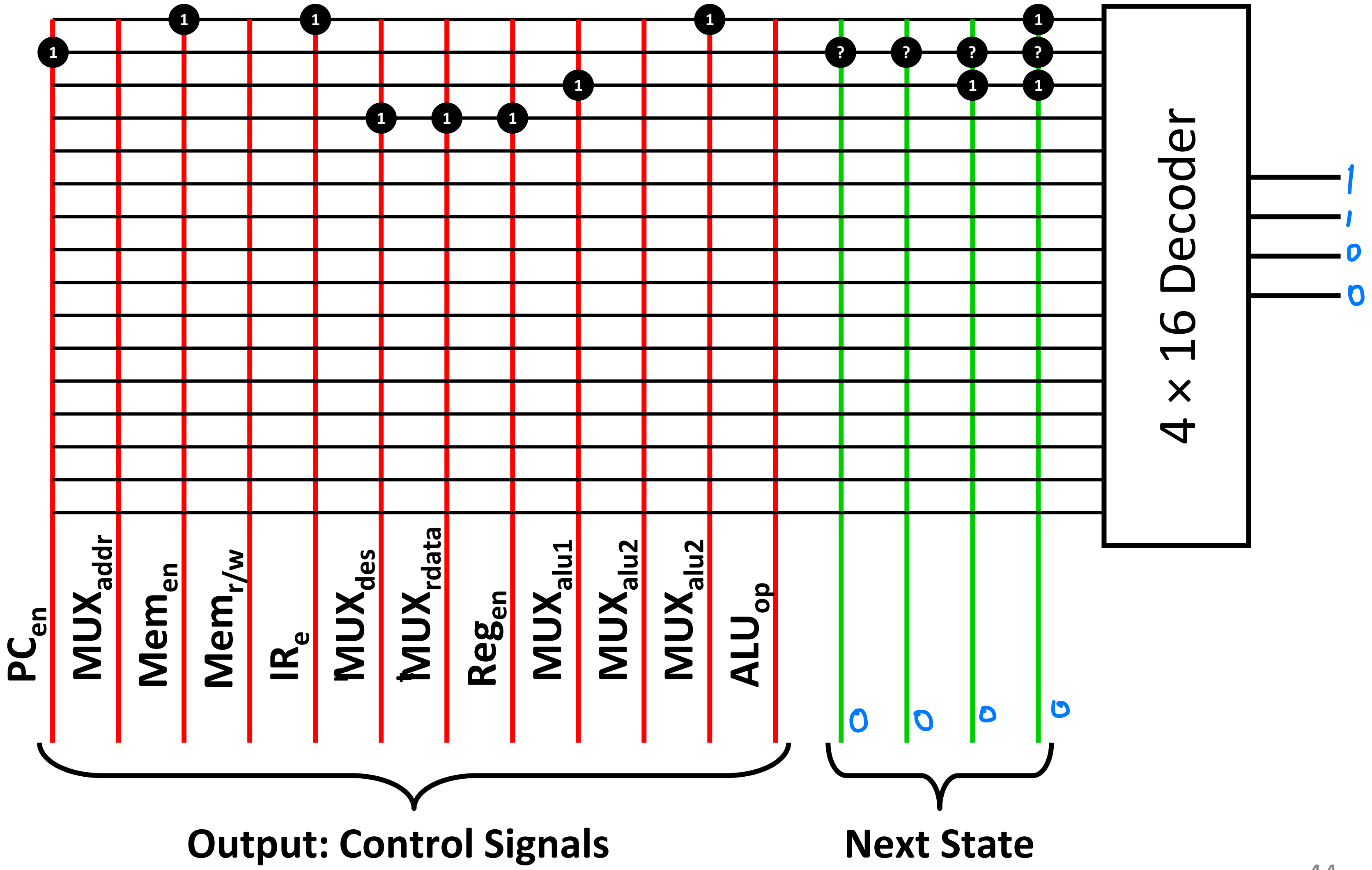


Add Cycle 4 (State 3) Operation

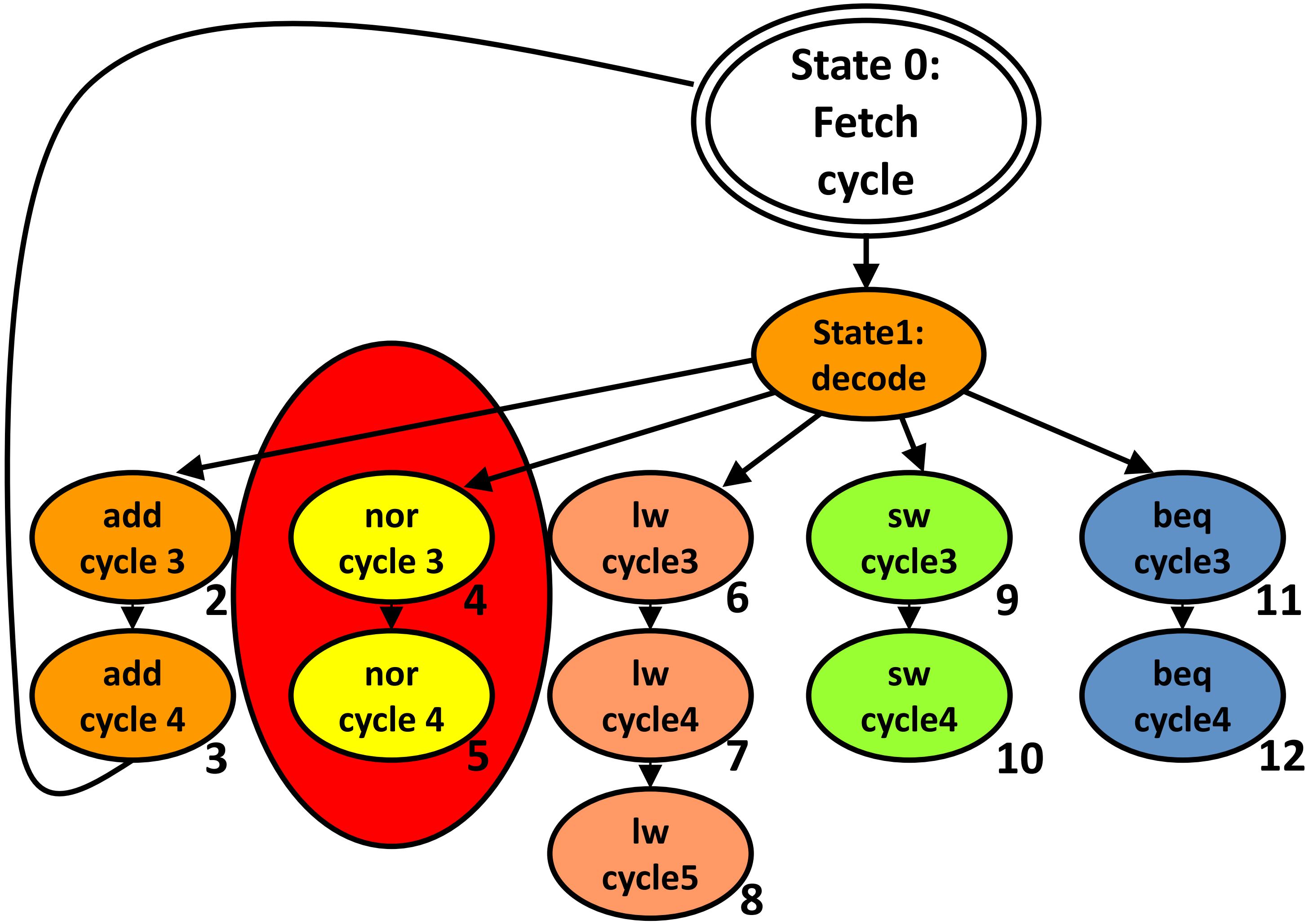
Send control signal to address MUX to select dest and to data MUX to select ALU output, then send write enable to register file.



Building the Control Rom

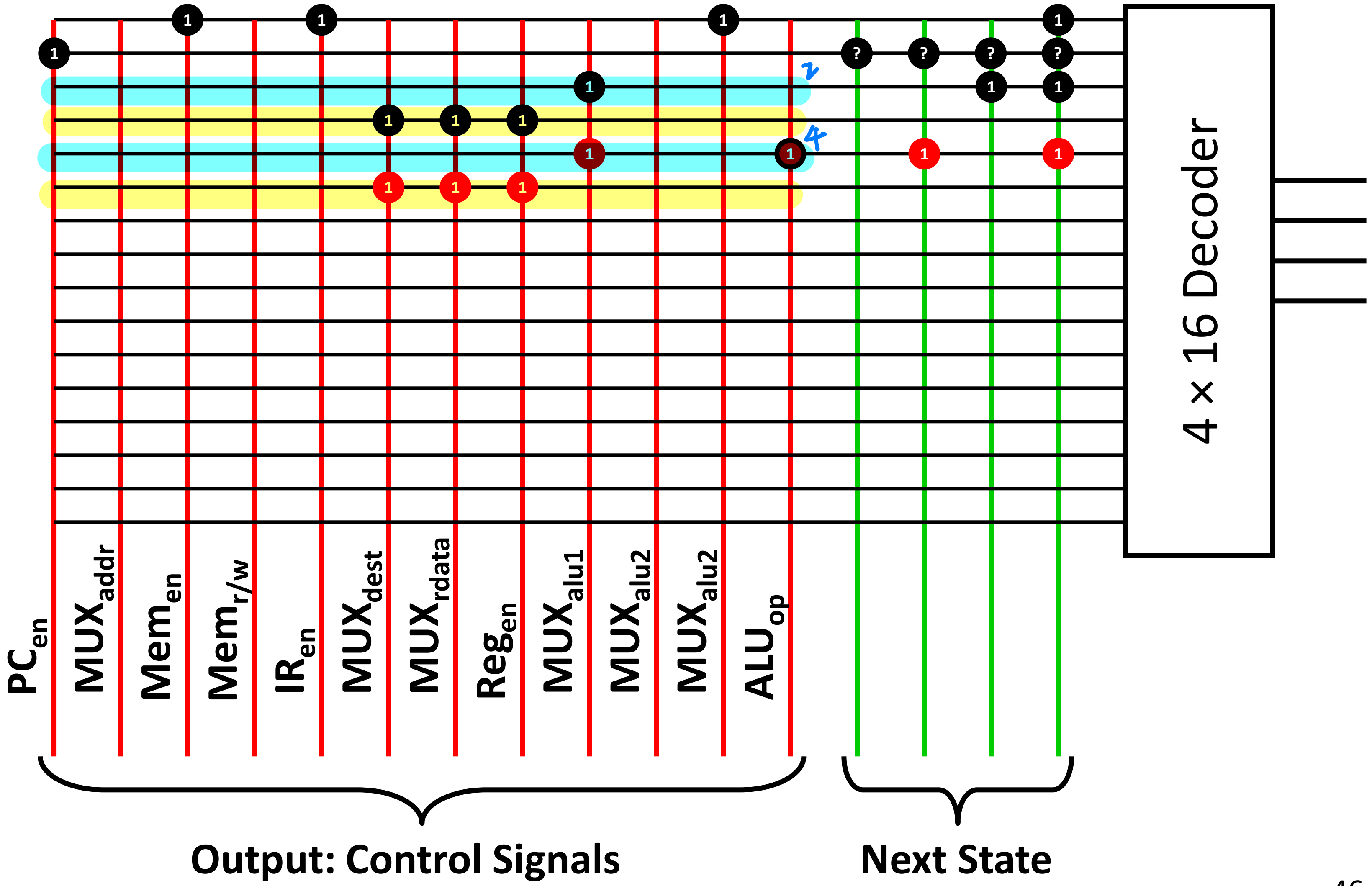


Return to State 0: Fetch cycle to execute the next instruction



Control Rom for nor (4 and 5)

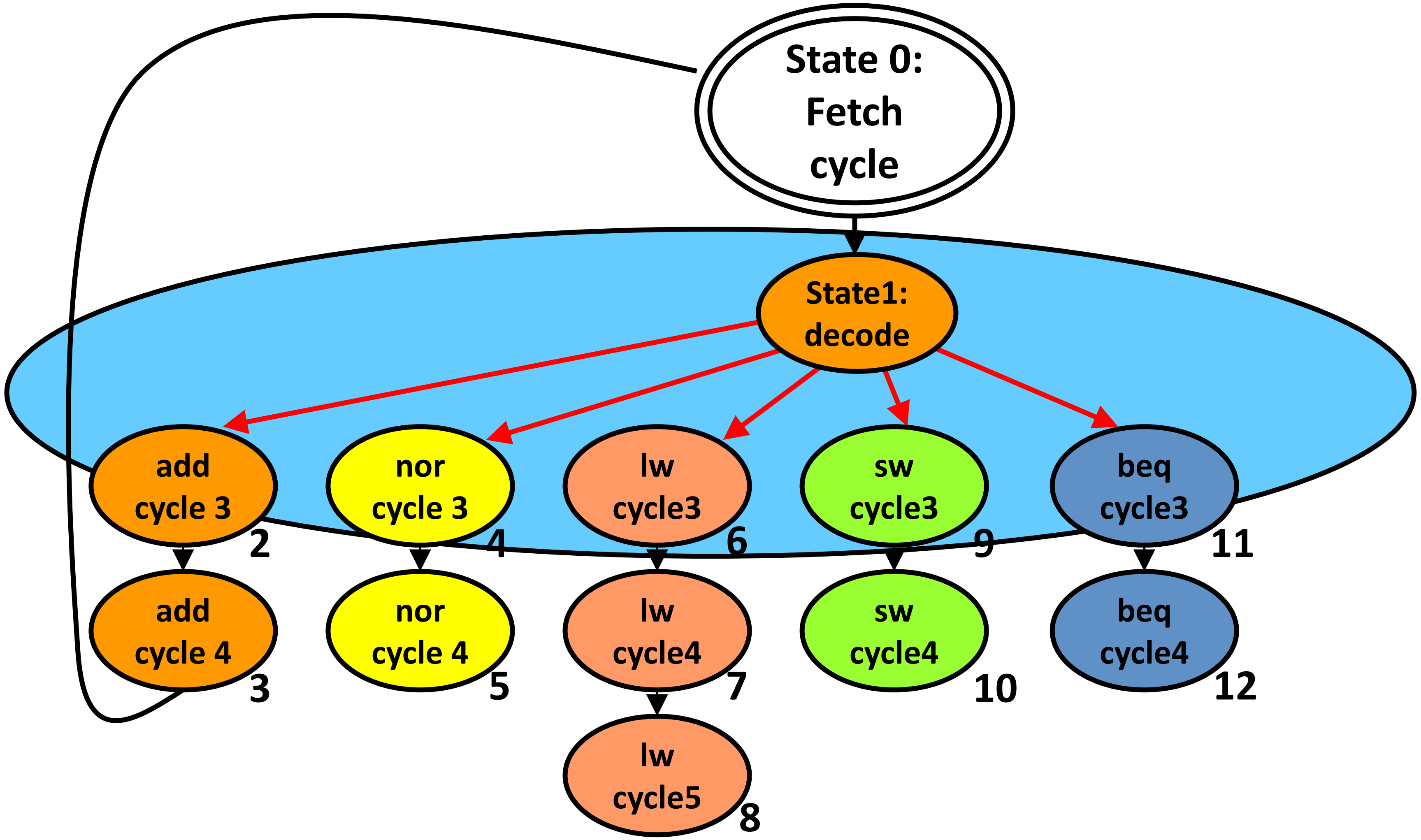
Same output as add except ALU_{op} and Next State



Agenda

- Short-comings of single-cycle
- Multi-cycle overview
- Fetching instructions
- Decoding instructions
- ADD/NOR execution
- **Choosing which state to transition to**
- LW/SW execution
- BEQ execution
- Performance

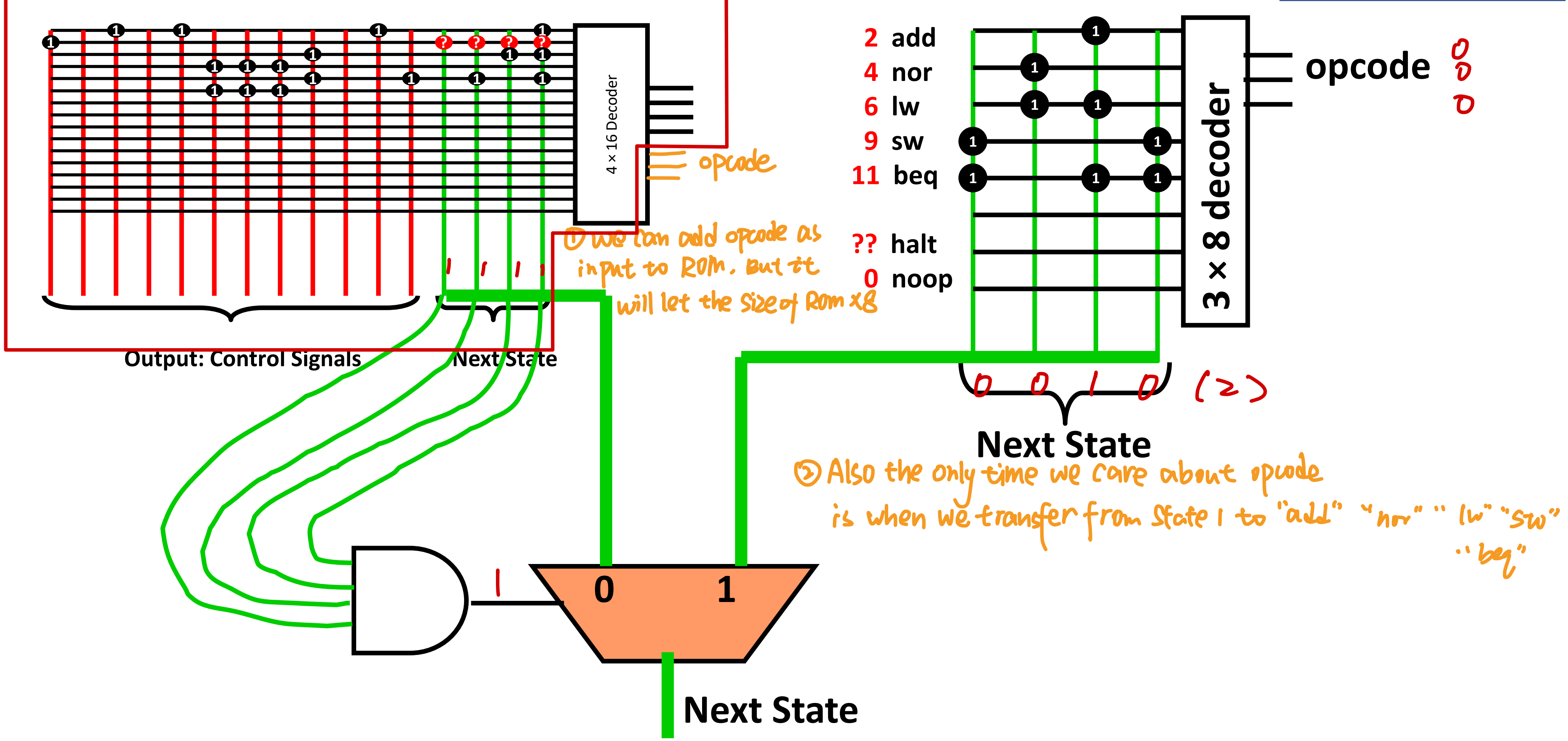
What about the transition from state 1?



Complete transition function circuit

Secondary ROM stores which state we should branch to after decode for each opcode

keep the control rom be the same.

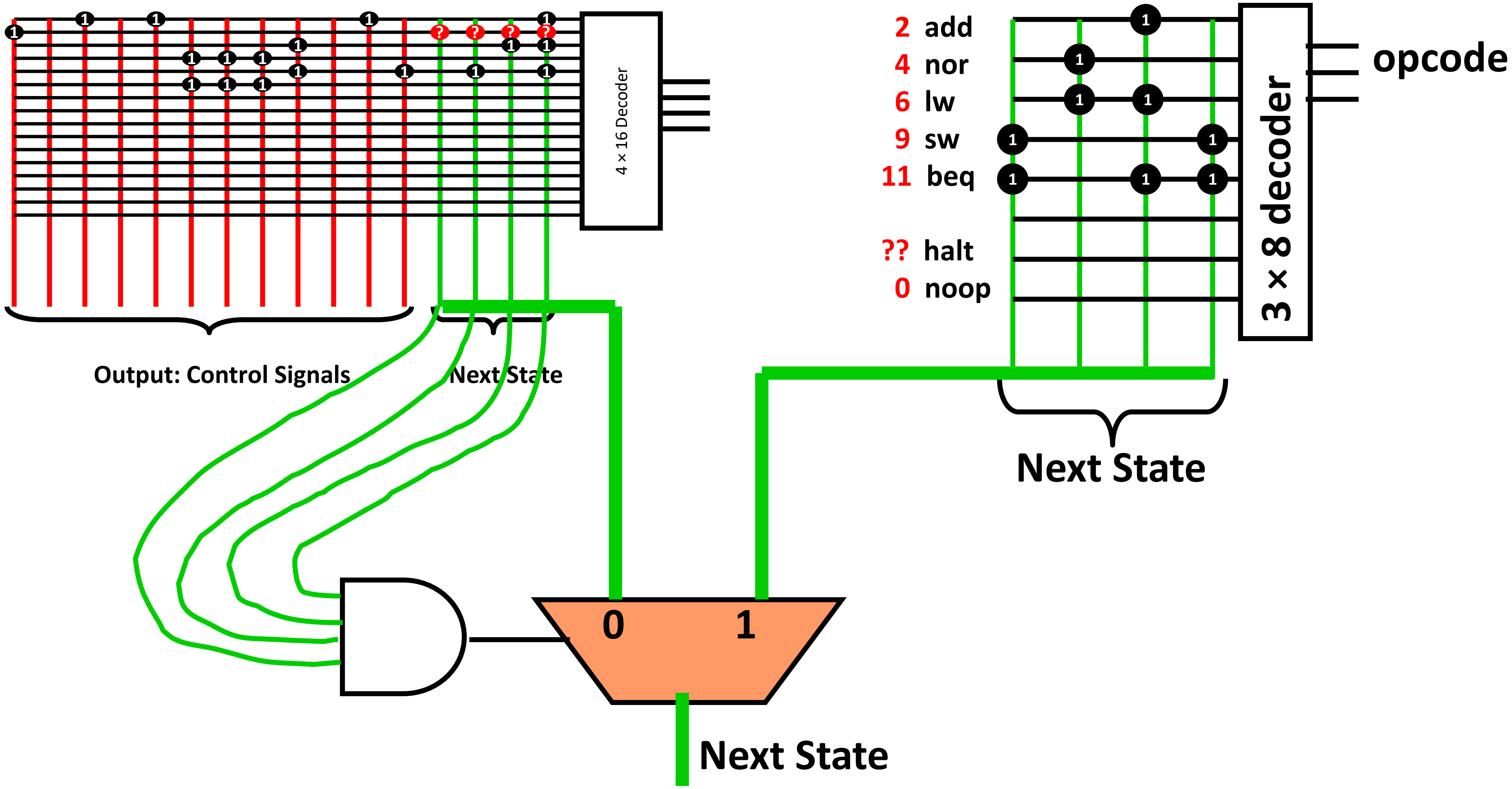


① we can add opcode as input to ROM, But it will let the size of ROM x8

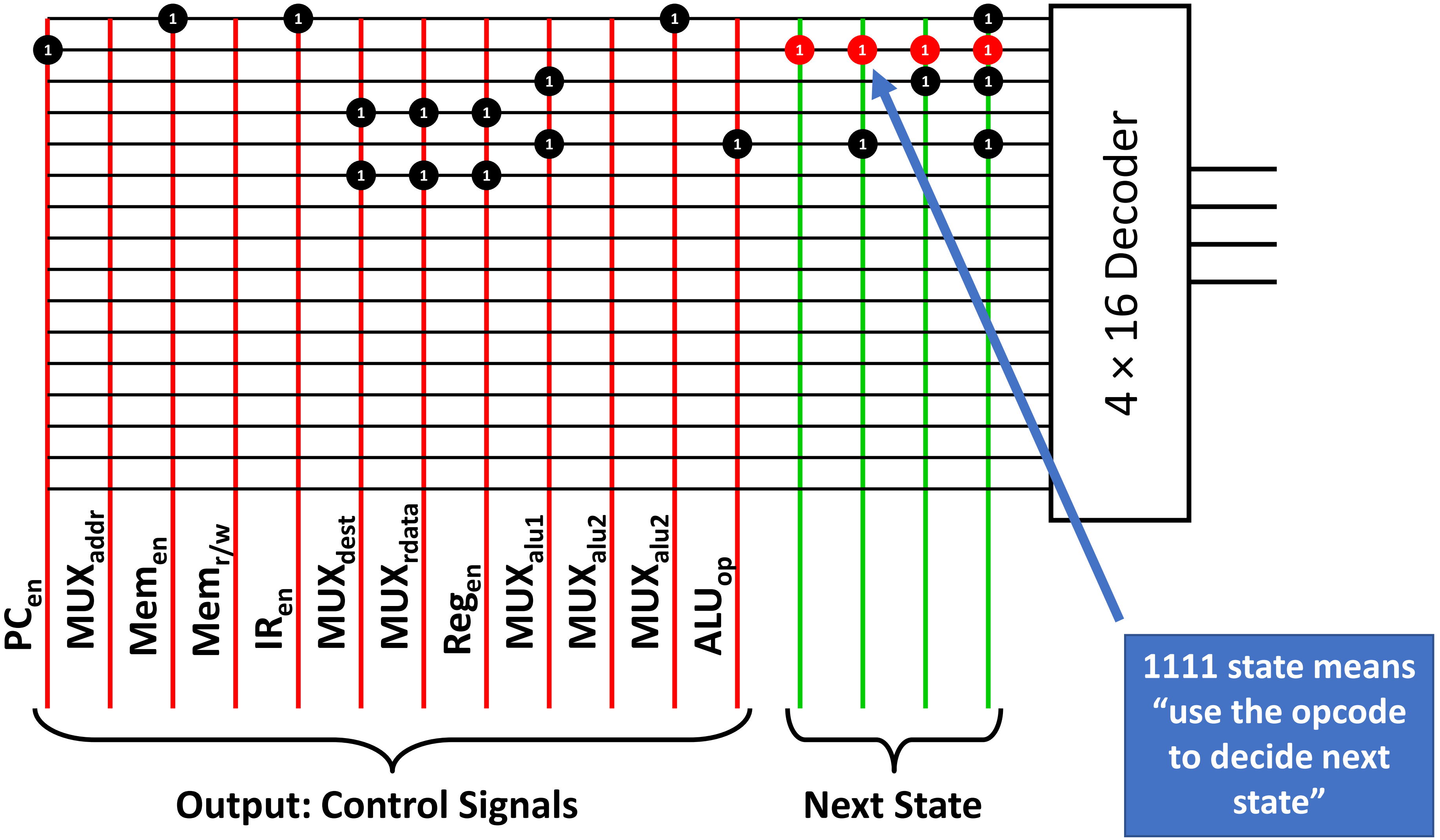
② Also the only time we care about opcode is when we transfer from state 1 to "add" "nor" " lw" "sw" "beq"

Complete transition function circuit

Secondary ROM stores which state we should branch to after decode for each opcode

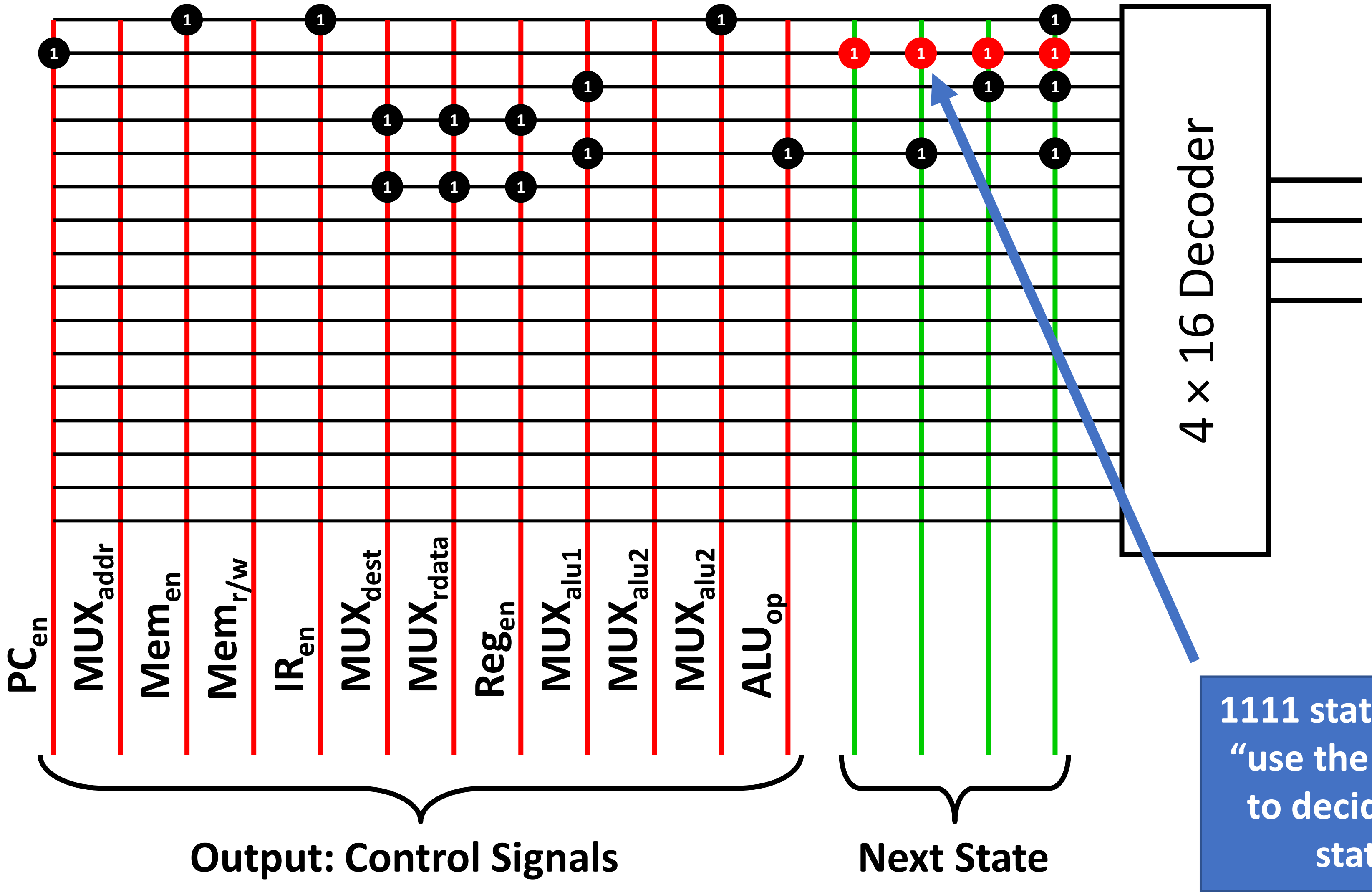


Control Rom (use of 1111 state)



1111 state means “use the opcode to decide next state”

Control Rom (use of 1111 state)



1111 state means “use the opcode to decide next state”